RADC-TR-87-222
Interim Report
July 1988

FOUNDATIONS OF ULYSSES: The Theory of Security

Odyssey Research Associates, Inc.

Daryl McCullough

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-87-222 has been reviewed and is approved for publication.

APPROVED: Joseph W. Frank

JOSEPH W. FRANK
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTC) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific doucment require that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A200 110

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-222

6a. NAME OF PERFORMING ORGANIZATION Odyssey Research Associates, Inc.	6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTC)
6c. ADDRESS *(City, State, and ZIP Code)* 1283 Trumansburg Road Ithaca NY 14850-1313		7b. ADDRESS *(City, State, and ZIP Code)* Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL *(If applicable)* COTC	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0098
8c. ADDRESS *(City, State, and ZIP Code)* Griffiss AFB NY 13441-5700		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
35167G	1065	01	02

11. TITLE *(Include Security Classification)*
FOUNDATIONS OF ULYSSES: THE THEORY OF SECURITY

12. PERSONAL AUTHOR(S)
Daryl McCullough

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Interim	FROM Apr 85 TO Apr 87	July 1988	146

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	02	
12	07	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Computer Security, ULYSSES
Hook-up secure, Theory of Security.

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This is an interim report for the Computer Security Properties Modeling Environment (ULYSSES) contract. This report begins by discussing the goals of the theory of security used by ULYSSES and how previous formulations of computer security failed to meet these goals. Next, ORA presents their theory of security, which incorporates a model of information flow with a model of processes as event systems. Included is a proof which shows that this definition of security, called hook-up security, is composable; if two processes are hook-up secure and they are hooked up in a secure fashion, then the resulting composite process is also hook-up secure. Finally, an illustration of the concept of hook-up security is presented by giving an example, a proof sketch that a simple process is hook-up secure.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Joseph W. Frank	22b. TELEPHONE *(Include Area Code)* (315) 330-3241	22c. OFFICE SYMBOL RADC (COTC)

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Acknowledgements

I would like to thank Ian Sutherland, Doug Weber, Stanley Perlo, Ram Varadarajanand, Tanya Korelsky, and Richard Platek for the discussions which generated and the ideas presented here. I would especially like to thank Tanya Korelsky, for her encouragement and suggestions, Stanley Perlo, for catching an amazing number of errors, and Toni Yengo, for all the pictures.

Daryl McCullough

Contents

Introduction

Ulysses is a system being developed at Odyssey Research in Ithaca, New York that is intended to assist in the design of secure computer systems. It will provide a rich environment in which previously defined secure systems and secure system components can be examined and incorporated into new system designs dynamically. This environment is based on the same principles of modularity and reusability that characterize modern programming environments and includes an automated theorem proving and verification engine.

This report discusses the foundations for the theory of security which will be used by Ulysses. The organization of the report is as follows:

- In Chapter 1, we discuss the goals for the theory of security used by Ulysses.
- In Chapter 2, we discuss some of the difficulties in applying previous models of security.
- In Chapter 3, we discuss the problem of Trojan Horses and covert channels, and how hooking up systems can make the problems worse.
- In Chapter 4, we discuss in more detail the Goguen–Meseguer approach to security.
- In Chapter 5, we briefly present Sutherland's model of information flow, which underlies the Ulysses definition of security.

- In Chapter 6 we describe informally the Ulysses model of systems as processes.

- In Chapter 7, we give a more formal model of processes as *event systems*, and show how Sutherland's information flow model is instantiated for event systems. We also give the motivation behind the definition of hook-up security, the definition of security used by Ulysses.

- In Chapter 8, we consider a sequence of candidate definitions of hook-up security, each correcting some of the flaws in the last definition. This sequence of properties lead up to a property which works as a definition of hook-up security.

- In Chapter 9, we prove a "hook-up theorem" which shows that when systems which meet our definition of security are hooked up in a legal manner, they form a composite system which is also secure by our definition.

- In Chapter 10 we sketch a proof that a small system, the delay queue, is hook-up secure.

- In Chapter 11, we discuss a formulation of hook-up security for state machines and prove a "hook-up theorem" analogous to that for event systems.

- Finally, in Chapter 12, we redo the proof of security of the delay queue using the state machine formalism.

Chapter 1

Requirements for a Model of Security

Secure design in Ulysses depends on flexible and sound theoretical foundations. To develop such foundations we examined previous formalisms for security, particularly the pioneering work of Bell and LaPadula in access control [BLP 76], and Goguen and Meseguer's noninterference model [Gog 84].

Our investigations convinced us that, for the purposes of secure design in Ulysses, these previous models of security were lacking in some respects. Some of the problems we found among these formalisms were:

- They were not based on observable behavior.
- They were not sufficiently implementation-independent.
- They could only be applied to completed systems, and therefore could not be used for the incremental development of a secure design.
- They only applied at one level of abstraction.
- They were only suitable for deterministic systems.

The biggest problem, however, was that there was no research on the interactions of trusted systems and processes—in particular, it was not known to what extent security was preserved when one connected several trusted systems into a distributed system.

The security formalism used by Ulysses is based on this previous work, but it goes beyond it in that it is intended to be useful in *design* as well as in *implementation*. The Ulysses security formalism can be used to analyze the security of isolated components and partially fleshed-out system designs, the implementations of which are still undetermined. This gives the designer greater flexibility, allowing him to

- reuse off-the-shelf secure components
- discover the security flaws of a design early so as to minimize wasted effort
- freely substitute components with equivalent security characteristics.

The definition of security in Ulysses was thus developed with the following desiderata in mind:

1. It should be *ostensible*.

   If a system is secure (or insecure), then that fact should be revealed by the behavior of the system and should not depend on details which have no observable consequences. We would like in the design process to specify the observable behavior of the system and prove once and for all that any implementation which reproduces that behavior is secure. This would give the maximum flexibility to the implementers of the design.

2. It should be sufficiently *general*.

   It should be useful across different levels of abstraction, from design to implementation. Since real systems are usually nondeterministic at some level, we would like to use the formalism to analyze both deterministic and nondeterministic systems.

3. It should be *context-independent.*

   Placing a trusted component into a new environment should not negate its security properties. This requirement is especially important for networks and distributed systems, in which trusted systems can be potentially connected in many different ways. The requirement that trusted systems be connectable into a trusted network is a very natural and important requirement, and was used as the basis for the Trusted Network Evaluation Criteria [Ars 85].

4. It should *preserve the advantages of previous security formalisms.*

   In particular, the definition of security should, like the Bell–LaPadula model, allow the use of some unverified "off–the–shelf" software for untrusted functions. It should also keep the intuitive connection between security and "preventing illegal information flow" found in the noninterference and information flow models.

In this report, we discuss these criteria and develop a theory of security to meet them. In the final chapters we prove the context-independence of our theory by showing that using our definition, security is preserved under the operation of connecting systems into networks, and we give a sketch of the proof of security for a simple system.

Chapter 2

Subtleties and Traps in Computer Security

The existing formalisms for computer security are tricky to use in real systems for two reasons:

1. The formal definitions of security do not always mean what we intuitively mean by security.

2. Existing specification and verification methodologies are not convenient for the proof or the statement of system security.

In the following, we elaborate on these difficulties.

Multi-level security requirements for a computer system are usually formalized in one of three ways:

1. As a requirement on access controls—The best-known example of such a requirement is given by the Bell-LaPadula security model [BLP 76]. The requirements defined by their model can be summarized as follows: Subjects (active entities such as users and processes) may only read

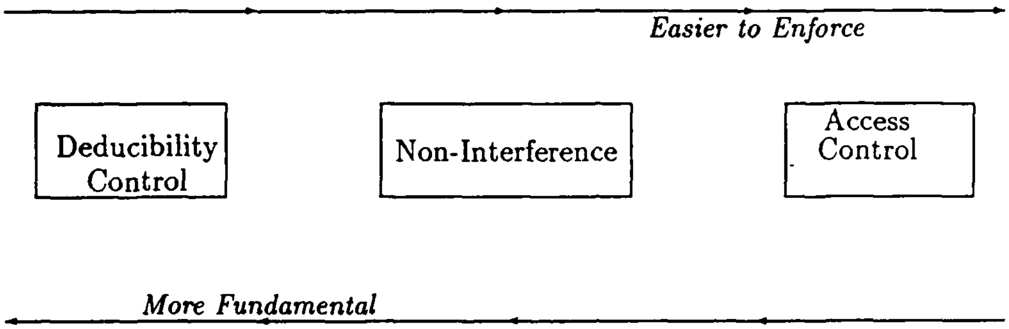

Figure 2.1: Trade-offs in Security Formalisms

objects (passive containers of information such as files) of lower level and may only write objects of higher level.

2. As a requirement on interference—The definitive statement of a requirement of this kind was made by Goguen and Meseguer [Gog 84]. Noninterference requires that the actions of users should only interfere with (or affect) what is seen by users of higher level.

3. As a requirement on deducibility—A clear statement of this requirement is given by Sutherland [Sut 86]. Essentially, the requirement is that users only be able to deduce the actions of users of lower level. This requirement is the definition of security that is implicitly assumed when encryption is used to protect data; although unauthorized users may read the encrypted data, it is assumed that they will be unable to deduce the meaning of the data.

The relationships among these formalizations is informally represented in figure 2.1 by two scales: *More Fundamental* versus *Easier to Enforce*.

The meaning of the phrase "more fundamental" lies in the relationship between the formal statement of the requirement and the informal statement of what is desired from the system. The purpose of security is to prevent information from falling into malicious hands, and deducibility control is a relatively straightforward formalization of that goal.

The meaning of "more enforceable" is that it is easier, in general, to see that a system obeys an access control requirement than a deducibility requirement. If a system has no access controls, then it is difficult to imagine how one could ever prove that it is secure.

Roughly speaking, access controls are a means of enforcing noninterference requirements and noninterference is a means of enforcing deducibility requirements. This might lead to the impression that access control is a stronger requirement than noninterference, which is a stronger requirement than deducibility control. In the way that these formalisms are actually used, however, this relationship is not true, for the following reasons:

1. Not all system interactions can be easily expressed as accesses. With access controls, it is easy to ignore such system variables as directories, the currently running process, timing information, etc.. Since such variables are not usually dignified with the label "object", information flow through them is not covered by access controls, and so such variables must be analyzed separately, as "covert channels". Thus, in actual practice, access controls do not insure noninterference or deducibility control.

2. With noninterference, if one uses a deterministic definition such as that of Goguen and Meseguer, and applies it to a nondeterministic system such as a network, the results may not have much to do with deducibility control.

As a compromise between having a security formalism which is too gross and one that is too difficult to use, we have chosen to use a nondeterministic form of noninterference. We were guided in the development of our formalism by examining a wide variety of pitfalls that await those who are not careful in their security work. Below are a few simple examples of such.

2.1 Is Nondeducibility the Same as Noninterference?

For nondeterministic systems, deducibility control is not the same as noninterference and is not necessarily in agreement with our intuitive notion of security. This is illustrated below by giving a system that is secure from the point of view of deducibility, but is intuitively insecure.

We will define a system to be multi-level deducibility secure if it is impossible during any run of the system for one user to deduce the actions made by another user during that run unless the first user has a higher current security level than the second. (This is an informal version of Sutherland's definition of information flow security. The formal statement is given in chapter 5.) This definition assumes that the system starts out empty of classified information and that the only way for classified information to enter the system is through the actions of users. From this viewpoint it is of primary importance to protect information about users' actions.

Deducibility security seems to be a reasonable criterion for security. To see if it really is adequate, though, we need to see how the criterion applies in sample cases.

2.1.1 Example: A Random Eavesdropping Machine

Consider a system $\mathcal{A}$ with two users, one of level *secret* and another of level *unclassified.* The *secret* user has a program which gives him text editing facilities for his *secret* files. We will assume that every input the *secret* user makes has level *secret.*

The *unclassified* user, besides text editing commands, has a special command "eavesdrop". This command has the effect that if the *secret* user has made any earlier *secret* inputs, then everything the *secret* user inputs after "eavesdrop" is echoed to the *unclassified* user's screen. If, however, there have been no earlier *secret* inputs then the system will generate random text to echo to the *unclassified* user.

Now, this system seems blatantly insecure, but nevertheless, there are no illegal information flows. This is because it is always consistent for the *unclassified* user to suppose that the *secret* user has made no inputs; it is always possible that everything the *unclassified* user sees following the "eavesdrop" command was created by the random text generator.

This example illustrates that deducibility security is not a very "robust" criterion for security; a slight change in the assumptions can lead to very different results. For example, if we assume that the *unclassified* user performs a statistical analysis of the sequence of bytes that he receives, then he can determine, with a high probability, which bytes convey real information and which are random. The bytes which are random will tend, with high probability, not to form coherent English sentences. Thus if a sequence of bytes reads "We attack at dawn", it is probably real and not random.

It is not immediately clear how we can distinguish between systems which are "robustly secure" and systems which are "fragilely secure", whose security depends on whether there is some kind of pattern to the high level inputs. While a full analysis of this subject may require a statistical or probabilistic treatment of information, we can at least say the following: If a system is robustly secure, then its security does not depend on the randomness of inputs; it should remain secure even when the high-level inputs are not random, but come in some kind of pattern.

One way that the inputs can have a pattern is if they come, not from a human being, but from another machine whose behavior is known. If we require that a secure system be deducibility secure for *any* choice of such a machine to provide the inputs, then perhaps the anomalous examples such as system $\mathcal{A}$ will be ruled out as insecure. This possibility is discussed in the next section.

2.2 Is Nondeducibility Preserved by Hook-Up?

Some formalizations for security have made implicit or explicit assumptions that the external interface of the system connects the system with flesh and blood human beings, when in actuality it is common these days for the system to be connected to other machines. The assumption that the system is complete is made explicitly in the Goguen-Meseguer model. In the Bell-LaPadula model, it is assumed that "reading" an object only transfers in one direction — to the subject doing the reading. In a network, however, all interactions, including requests to read, can potentially transfer information in both directions.

It is commonly believed that if we take into account the two-way information transfer of protocols, that it is enough to connect secure machines to secure machines in a secure way to get a secure network. This is the assumption behind the DOD guidelines for network security [Ars 85]. However, our next example shows that unless security is defined carefully, it is possible that hooking together secure machines will produce an insecure network.

For illustration, we consider once again the system $\mathcal{A}$ from section 2.1.

2.2.1 Example: Hook-Up Destroys Nondeducibility

Although there are no illegal information flows for system $\mathcal{A}$ in the environment in which the only external interfaces connect the system with users, we shall see that illegal information flows appear when it interfaces with another system.

Let system $\mathcal{B}$ be a second system which has also two levels, *secret* and *unclassified.* In this system, the *unclassified* user and the *secret* user both have text editing facilities, which we will assume work by echoing everything typed by each user to his own screen. In addition, every input made by the *unclassified* user is echoed to the *secret* user's screen *before* it is echoed to the *unclassified* user's screen. There are no illegal information flows on this system, either, since the *secret* user's inputs have no effect on the *unclassified* user.

Now, consider what happens when we connect system $\mathcal{A}$ with system $\mathcal{B}$—we imagine connecting them so that the *unclassified* outputs of each system become *unclassified* inputs to the other system. We also allow both systems to receive *unclassified* inputs from an *unclassified* user's keyboard, and to receive *secret* inputs from a *secret* user's keyboard, and we allow the *unclassified* outputs of either system to be echoed on the *unclassified* user's screen, and likewise the *secret* outputs of either system to be echoed to the *secret* user's screen.

When the *unclassified* user types in anything at all, the inputs are sent to system $\mathcal{B}$, which then echoes it twice; once as a *secret* output to the *secret* user's screen and system $\mathcal{A}$, and again as an *unclassified* output to the *unclassified* user's screen and system $\mathcal{A}$. When the *unclassified* user receives the echo from system $\mathcal{B}$, he knows that there was a previous echo of a *secret* signal to system $\mathcal{A}$. He can then issue the command "eavesdrop" and know for certain that it will work. Therefore, with 100 percent confidence, the *unclassified* user can know that he will receive every input the *secret* user makes to system $\mathcal{A}$ (and not some randomly generated text).

Thus we see that knowing that a system is deducibility secure is not enough to know that it will still be secure when connected to other secure systems. Since we want to allow multi-level secure systems to be connected to each other, we need a definition of "hook-up" security that tells us under what circumstances this can be done safely. We will delay providing this definition until we give a few more examples of the problems with security specifications.

2.3 The Dangers of Underspecification and Nondeterminism

Now we turn to a new example which illustrates that even when we have a good formal definition of security, there are problems in knowing when a system meets the definition. We need to first have a short digression on what it means for a system to meet various kinds of requirements.

2.3.1 Liveness and Safety

Properties of a computer program or system are sometimes divided into "safety" and "liveness" properties. Safety properties are often described as "negative" requirements: they require that certain behaviors of the systeme *never* occur. They are statements of the form, "Whatever the system does, it must not do such-and-such." For example, the partial correctness of a sort routine can be stated as "Whatever the routine does, it must *not* terminate with the list unsorted."[1] If a sort routine is demonstrated to meet this partial-correctness condition, then it does not necessarily mean that the program will sort a list, only that *if* it halts, then the list will be sorted; it is possible for a partially correct sort routine to run forever and never give any result.

Liveness properties are "positive" requirements, stating that certain behavior of the system *must occur.* Liveness properties include statements of the form "This action must be performed eventually" or "This action must be performed infinitely often". Fair scheduling is an example of a liveness property: it requires that every process *must* be scheduled.

Safety properties are much more manageable, both for the designer and for the programmer. The designer gives the requirement, and the programmer has the freedom to do anything at all as long as the requirement is met. Thus in general the implementation is more definite and more deterministic than the design, since choices made by the implementer constrict the behavior of

[1] The specification of partial correctness for a computer program S can be written in the form $\{P\}S\{Q\}$, where P is an assertion about the state of the system before executing program S, and Q is an assertion about the state of the program after the program S terminates normally (that is, without raising any error conditions). To prove that S is partially correct with respect to the specification $\{P\}S\{Q\}$, it is not necessary to prove that S will ever terminate; S may go on computing forever or until it runs out of memory and raises an error; however, if, after starting in a state of the system in which P is true, one executes S and it does terminate, then Q will be true.

the system. As far as safety properties are concerned, if one implementation is correct, then a more deterministic system (one with fewer possible behaviors) is also correct.

For liveness properties, however, the relationship between implementation and design is not so straightforward. If one system meets a liveness requirement, it does not necessarily follow that a more deterministic system will meet the requirement. Therefore, when there are liveness requirements, one must be much more careful that implementation decisions do not interfere with the requirements.

Consider, for example, the liveness requirement that a scheduler be fair. A nondeterministic implementation of such a scheduler might be one which randomly chooses which process to schedule next in such a way that with virtual certainty each process is scheduled eventually. However, if the scheduler is made more deterministic, by having it make the choice the same way each time, then this modified scheduler is not fair.

2.3.2 Is Security a Safety Property?

It is sometimes claimed that security is a safety property, that it is sufficient to give the requirements at the design level and leave the rest up to implementers. The reasoning goes that, since a system which does nothing is secure (there are no illegal information flows in such a system), it must be that security does not require that anything positive happen.

This reasoning is not correct; the fact is that as long as the design does not uniquely specify what happens on the system, it is possible for an implementation to correctly meet the design and still not be secure. For such designs, therefore, security cannot be exclusively a safety property. Our next example provides an illustration.

2.3.3 Example: Resource Sharing and Underspecification

This example should be familiar to those who have studied actual attempts at breaking systems. We imagine a system which obeys a Bell-LaPadula definition of security—each process has a level, and is allowed to only read files of lower level and write files of higher level. We will suppose that:

- The system only supports one official programming language.
- The operational semantics of this language has been specified.
- The compiler has been verified to respect these semantics.

In the language specifications, the value of an array component is specified to be equal to the last value read into it, if any; otherwise the value is undefined. In the implementation of the programming language, when a program declares an array of a certain size, the operating system sets aside a chunk of memory large enough to hold that array.

Now, suppose that an *unclassified* user declares an array of bytes as large as all of memory, and then prints it out without assigning any values to it. Technically, what gets printed out is undefined, but it happens to look just like whatever used to be in memory. If the user has just logged on, then the chances are that what is lying around in memory is the password table which the operating system had to load for the logon. Now, did this system obey the specification, or didn't it? If we interpret "undefined" and "undetermined" as meaning essentially the same thing, and if we interpret security as a safety property, then we are forced to say that this was a correct implementation; everything required (that is, determined) by the design is actually met by the implementation.

The problem is that even though, at the design level, certain results are undefined or undetermined, at the implementation level these results are determined by lower level facts; and the precise way in which the nondeterminism is resolved may provide an illegal channel which does not exist at the design level. What is needed is a definition of security at the design level which will insure that any correct implementation is also secure.

2.4 Manifestly Secure Systems

It may be argued at this point that nondeterminism is the source of all the problems—that it is meaningless to talk about the security at the design level unless the design completely determines what happens on the system. If this is true, it is a sad fact indeed, since it implies that the time-honored tradition of top-down development, with large decisions made by the designers and the details worked out by the implementers, is not workable for security. It implies that the security work can only begin when the system is finished.

It is our opinion that design-level security is not meaningless; it is just difficult, or at least subtle. There are, nevertheless, trivial examples which illustrate that it is possible, at the design level, to insure that a system is secure without completely determining the system. We give an example pictorially in figure 2.2.

In the figure 2.2, the boxes represent machines which are components of a larger system, and the arrows represent one-way (or "half-duplex") communications lines between machines. The above picture describes a distributed system made up of four components, two *unclassified* machines and two *secret* machines. The *secret* machines are connected, and the *unclassified* machines are connected, but the only connection between components of different levels is a one-way connection from an *unclassified* component to a *secret* component. This system is "manifestly secure", even though we have specified nothing at all about how the two systems behave.

The important feature about the design of secure systems is the *connectivity*, the way that system components are allowed to communicate. If the connectivity is manifestly secure, then any implementation that correctly maintains the connectivity is secure.

For security, we can say informally what it means to correctly implement a

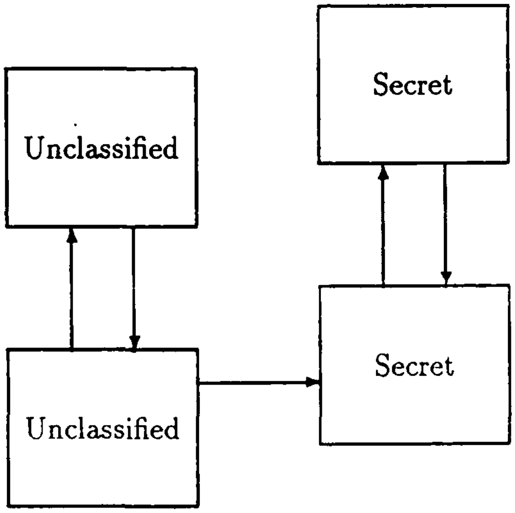

Figure 2.2: Manifestly secure system

design in terms of connectivity.

1. For each component of the system, the corresponding component of the implementation must meet all the design requirements for that component. These requirements do not necessarily specify the actions of the component completely.

2. The interfaces between components must meet all the requirements of the design.

3. The implementation must contain no new connections between components which are not present in the design.

Point 2 is intended to insure that whatever nondeterminism is present in the specification of the design is not resolved on the basis of any information other than that explicitly available through the connections present in the design. In the example in section 2.3.3, we can see in retrospect that at the design level we failed to specify what "communications lines" would be available to *unclassified* processes. If we had introduced at the design level the connections linking components, we would have seen that the memory management process necessarily has lines linking it with every other process, as shown in figure 2.3. Unless either memory management or the *unclassified* process is more completely specified, the system can be seen to have insecure connections, since information can flow from the *secret* process to the *unclassified* process through memory management.

The additional requirement on memory management that is necessary is that actions of *secret* processes should not interfere with the inputs to the *unclassified* processes, which is obviously not satisfied if the *unclassified* process can read memory that was written by *secret* processes. Viewed in this context, the memory management is a "trusted process", since it involves processing at several security levels at once. For such a process, it is necessary to have a theory of "hook-up security" such that it can be connected to other processes which are either "manifestly" or "hook-up" secure to yield a secure system. The theory of "hook-up security" will be developed in chapter 9.

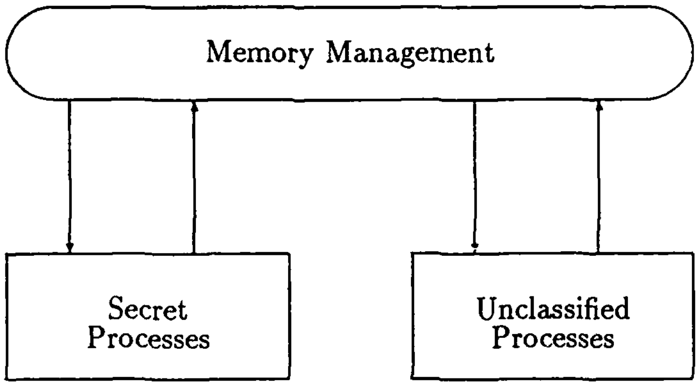

Figure 2.3: Information flow for resource sharing

In conclusion, we hope that the examples have shown some of the subtleties of computer security. Keeping these subtleties in mind, the authors are working towards a computer security formalism and methodology with the following properties:

- The definition of security must imply the fundamental definition in *terms of deducibility control.*
- The security of a complex system should follow from the security of the components and their interconnections. Thus, a network of secure machines should be secure.
- A proof of security at the design level should guarantee the security of any correct implementation of the design.
- It is possible to say at the design level that a system is secure, and still have large portions of the system nondeterministic and unspecified.
- The methodology for proving security must be able to make use of the connectivity of complex systems. That is, security should be decomposable, so that security can be insured by guaranteeing that the

components are correct, and that the interfaces between components and to the outside world are correct.

Chapter 3

Covert Channels and Degrees of Insecurity

In this chapter we will investigate various kinds of covert channels, of various degrees of insecurity, and the ways in which the different kinds can be interconverted and combined.

We will distinguish between two kinds of insecure systems. The first kind contains some security loophole or trap door which allows the spy to bypass normal access controls and to directly receive classified data. The Bell-LaPadula security model [BLP 76] is intended to prevent this kind of insecurity; for a system to be secure in the sense of Bell-LaPadula, every possible sequence of system state transitions must result in a secure state; i.e., one in which no user has access to classified data unless he is officially authorized to have that access.

The second kind of insecure system is one which disallows *direct* access of data by unauthorized users, but nevertheless allows for *covert channels*. A covert channel is an indirect communication path between users. For the second kind of insecure system, it is often necessary for the spy to have a partner which is privileged to see classified data and can signal this information to the spy. The partner in high places can either be a human (in which case it is unnecessary to communicate through the system at all; the two can pass notes in the cafeteria) or a *Trojan horse* program. An important thing to realize about the Trojan horse program is that it does not necessarily violate any rules of security; it may even be *proven* to be secure according to some formal model of security. However, if the system on which the program is running is insecure, it may be possible for the Trojan horse to communicate classified information to the spy through the side effects of perfectly normal innocuous operations. The prevention of low-level side effects of the behavior of high-level programs is the goal of various versions of noninterference requirements on systems, including the Goguen-Meseguer noninterference property[Gog 84] and the Bell-LaPadula "star" property (which disallow the writing of low-level data by high-level programs.)

3.1 Kinds of Covert Channels

A covert channel can be characterized by how much information can be sent over it, how dependable it is, and the degree of coordination needed between the sender (the Trojan horse) and the receiver (the spy). A complete characterization would include a probabilistic treatment of the dependability. Here we will treat dependability in an especially simple way; we will assume that information is always either trustworthy or not. Thus in trying to determine the answer to a yes-no question, a spy can get three answers: "yes", "no", or "maybe" (rather than "yes with a 30% probability").

3.1.1 Conventions and Notations for Systems

To illustrate the possible behavior of systems, let us introduce a pictorial notation for the traces, or possible histories, of systems. We depict a trace of a system by giving a timeline running vertically, with the future of the system toward the top and with the past of the system toward the bottom. Horizontal vectors directed toward or away from the time line of a system represent inputs to and outputs from that system, respectively. We will use unbroken lines to represent *unclassified* inputs and outputs, and wavy lines to represent *secret* inputs and outputs. In figure 3.1, we show a trace in which there is a *secret* input followed later by an *unclassified* output.

In this discussion, we consider only ***input total*** systems; any input must be accepted by a system. The reason for this assumption is that it is then easier to analyze information flow: information can only enter a system through input events, and can only leave a system through output events.

To represent two systems operating in parallel, we show their timelines together as in figure 3.2, which shows a trace in which an *unclassified* internal signal is sent from system $\mathcal{A}$ to system $\mathcal{B}$, followed by an *unclassified* input to system $\mathcal{A}$, followed by an *unclassified* output from system $\mathcal{A}$, followed by an *unclassified* input and output of system $\mathcal{B}$. The following is a list of notes about conventions for parallel composition:

- Events common to the two component systems are internal, and must be an input event for one system and output event for the other system, and must have the same security level for the two systems.
- We will use the convention that events common to two systems will be shown as vectors pointing right out of the left timeline or left out of the right timeline.
- We assume that there is no propagation delay for exchanged signals; the output from one system is simultaneously an input to the other system.
- An input of a combined system is any input to either system which is not an output of the other system. Thus such inputs will be shown coming from the left toward the left timeline, or from the right toward the right timeline.
- An output of the combined system is an output of either component system which is not an input of the other system. Thus an output for the combined system will be shown as a left-pointing vector from the left timeline, or as a right-pointing vector from the right timeline.
- The legal traces of the combined system are traces such that the events local to each component system form a legal trace for that system.

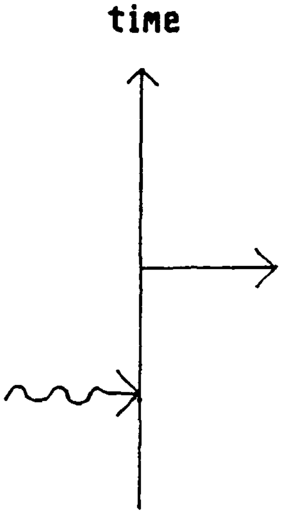

Figure 3.1: Pictorial Representation of a Trace

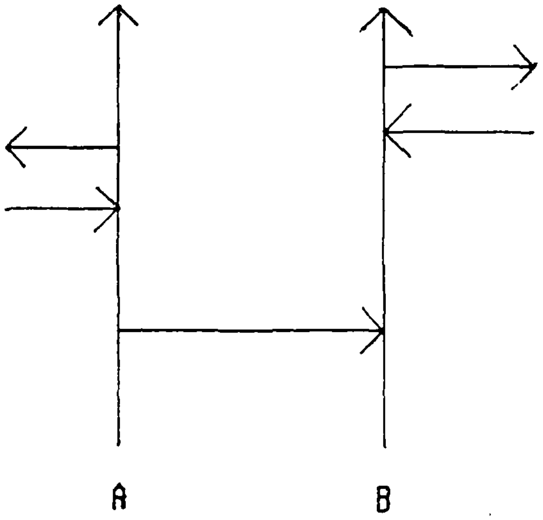

Figure 3.2: Hooking Up Two Systems in Parallel

- We are assuming *no* relationship between the processing rates of two systems in parallel—any number of outputs of one system can take place in the time between two outputs of the other system.

3.1.2 One-Bit Channels

Since any kind of information can be encoded as a sequence of bits (each bit being one of two possibilities), it is only necessary for a Trojan horse to be successful that it be able to communicate two different messages, as long as it is possible to create arbitrarily long sequences of alternations of the two messages. We will call a covert channel a *one-bit channel* if it is possible using the channel to communicate two different messages.

The situation in which a Trojan horse is communicating with a low-level user can be pictured as in figure 3.3. In this set-up, there are three data-processing components: the Trojan horse, the leaky system, and the spy. To communicate, there need to be two distinguished behaviors of the Trojan horse, which we will assume, for purposes of illustration, to be particularly simple:

- send signal a to the system

- send signal b to the system

For this set of behaviors for the Trojan horse, a system allows a one-bit channel if it responds to the classified a from the Trojan horse in one way, say by outputting "yes" to the spy, and responds to signal b another way, such as outputting "no". A system with a one-bit channel is not quite as bad as one which allows the direct reading of classified data by the spy, because it requires cooperation by a Trojan horse. However, given that there might be a Trojan horse willing to cooperate, a one-bit channel is pretty bad for security; with time, any amount of information can be transmitted.

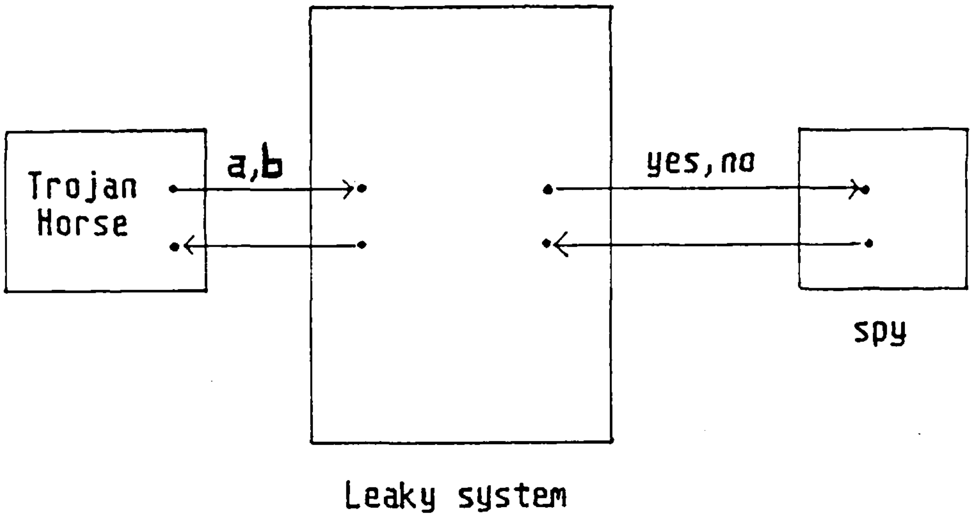

Figure 3.3: A Trojan Horse, a Leaky System, and a Spy

3.1.3 Noisy One-Bit Channels

An insecure system has a noisy one-bit channel if the signal from the Trojan horse is sometimes ambiguous: that is, if sometimes from the response of the system it is impossible to tell which of two alternative signals was received by the system from the Trojan horse. An example of this situation is a slight modification of the one-bit example above. The Trojan horse's input of an a can give rise to two possible responses of the system: "yes" or "maybe". Likewise, b can give rise to two possible responses; "no" or "maybe". The spy must ignore "maybes", since they are unreliable. Although communication through a noisy channel is slower than through a clear channel, it is still possible to communicate any amount of information accurately; the Trojan horse only needs to repeat his signal often enough, until the response is something other than "maybe".

The possible histories of the system are indicated schematically in figure 3.4.

3.1.4 Half-Bit Channels

The two kinds of channels described above are clearly undesirable (to security engineers; they are just as clearly desirable for spies). The existence of such channels can only be tolerated if

- The channel is noisy enough that no appreciable information will be transmitted in a reasonable amount of time.

  The problem with this criterion is that it is often difficult to estimate the capacities of channels. It is also difficult to come up with an objective notion of what is an acceptable rate of information leakage. Is one bit per hour acceptable? If such a channel were used to communicate passwords to the spy, then the spy could possibly obtain a ten-byte password in just eighty hours, or a little more than three days. (Passwords are not customarily changed that often.)

- The security engineer has confidence that there are no Trojan horse programs which are allowed to access classified data.

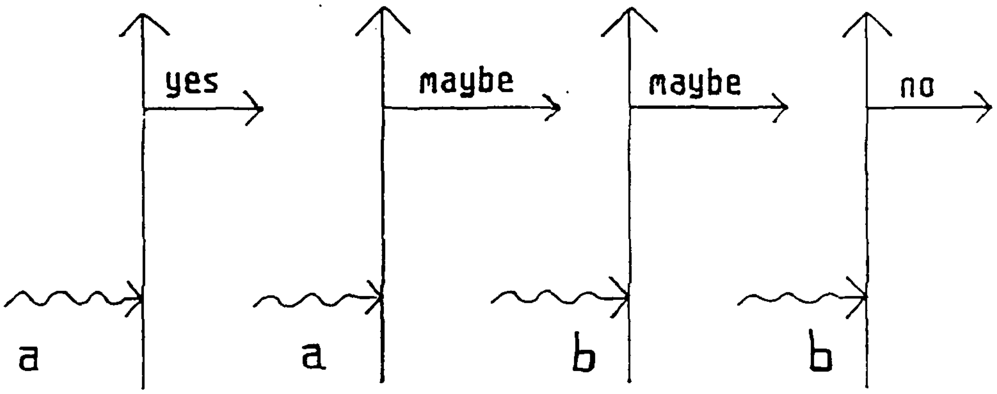

Figure 3.4: Histories of a Noisy One-Bit Channel

> Without careful analysis, such confidence is surely unwarranted. As I pointed out earlier, the Trojan horse program does absolutely nothing illegal; it lives within the rules of the game, only sending classified information to authorized users or "trusted" software such as the system in question. The Trojan horse program may even be provably secure. It is also conceivable that a piece of software may act like a Trojan horse even when that was not the intention of the programmer who wrote the software.

If one bit channels are clearly unacceptable, then are there other "almost secure" systems which a security engineer could live with? Next we consider two less useful covert channels, which could be called "half-bit channels". For these channels, the Trojan horse does not have the choice of two different signals; it has a single signal, and it can either choose to send it or not.

Positive Half-Bit Channels We will call a covert channel a "positive half-bit channel" if the spy can trust a "yes" response of the system to indicate that the Trojan horse sent a signal, but there is no indication that it did *not* send a signal. The possible histories of such a system are given by figure 3.5. If the system receives a classified signal then it may respond with either "yes", or "maybe". If the system does not receive a signal, it responds "maybe". The spy, seeing "yes", knows that the Trojan horse has signaled him, but seeing a "maybe" cannot be sure.

If the spy ignores all "maybes" as unreliable, then the communication he receives from the Trojan horse is singularly boring; an unbroken string of "yeses". He can infer some information from this string; for instance, if he counts them, he can place a lower bound on the number of signals the Trojan horse actually sent. However, since some signals may have resulted in "maybes", he cannot depend on the exact count.

This half-bit channel seems like a rather poor way to spy, and some might argue that it is not worthwhile to bother eliminating all such puny channels. Yet in certain cases it is wise to be paranoid. An example of when such a channel would be very useful to a spy is easy to cook up. Suppose that

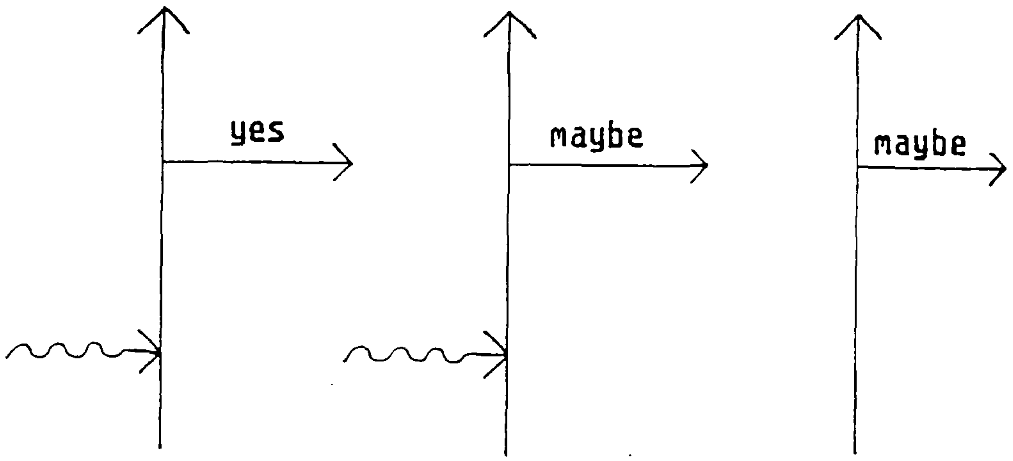

Figure 3.5: Histories of a Positive Half-Bit Channel

a Trojan horse were to be placed into the computer system of a gigantic corporation. The Trojan horse could read classified files and look for evidence of something big coming up—a stock split, or a merger—that would cause the value of the company's stock to shoot up overnight. In this case, for the Trojan horse to be of value to the spy, it is enough that it be able to communicate a single word "yes" (meaning "buy stock now!").

Negative Half-Bit Channels For those who are worried about positive half-bit channels, we will introduce yet a poorer channel, which they may be willing to tolerate. In this case, the system can indicate to the spy that there has definitely *not* been a signal from the Trojan horse, but there is no way to indicate that there definitely has been a signal. We call this case a negative half-bit channel, since it can only be used to give negative information.

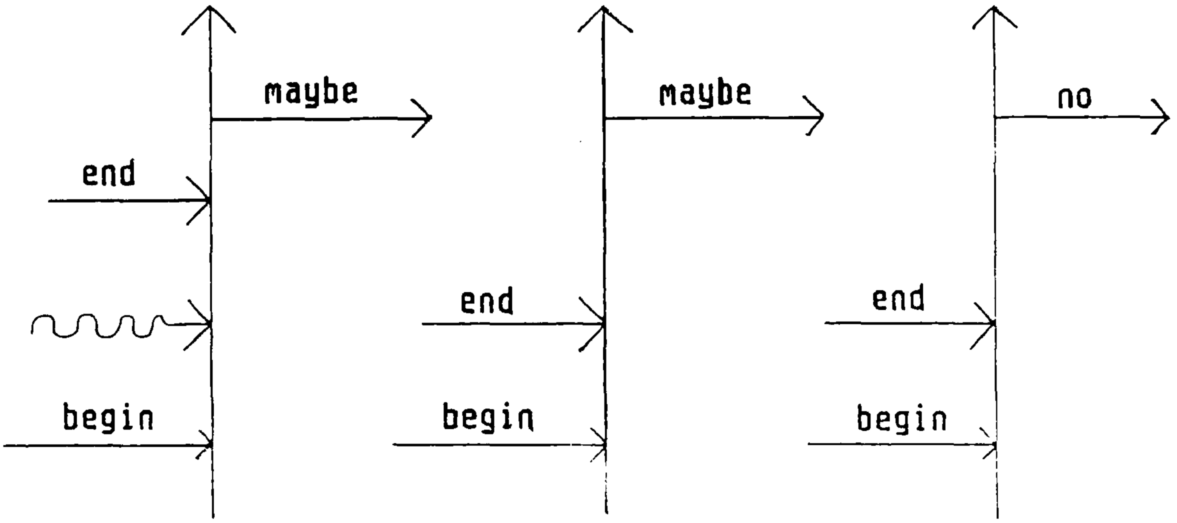

Figure 3.6: Histories of a Negative Half-Bit Channel

The possible histories for such a channel are given in figure 3.6. For this channel to work, unlike the other channels which allowed the spy to be a passive observer, the spy must periodically request information from the system. The spy starts an information period with the "begin" request, and ends it with the "end" request. If in the period between "begin" and "end" there has been no signal from the Trojan horse, the system responds "no" or "maybe". If there has been a signal, the system responds "maybe". Once again "maybes" are ambiguous; and if the spy ignores them, he is left with an unbroken string of "nos". What can he learn from such a string? It seems that the answer is : absolutely nothing! He cannot depend on the "nos" to indicate that the Trojan horse did not wish to signal him, since it is possible that it had tried to signal, but produced "maybes"; or it could be that the Trojan horse wished to signal, but had not gotten around to it yet. (The Trojan horse after all might have other duties, such as acting like a word-processor or a chess program.)

Because the spy is unable from his observations to deduce anything about the intentions of the Trojan horse program, a system with only this kind of covert channel could be called "deducibility-secure". It also obeys a kind of noninterference condition; it is not possible through this system for a Trojan horse to reliably affect what the spy sees. There is no way for it to force a "no" to be produced by the system; if it does nothing, it is possible for "maybe" to be produced instead of "no". There is also no way to force "no" *not* to be produced, because it is always possible that a pair of requests from the spy will come so close together that there is no time for the Trojan horse to send a signal. Likewise, the Trojan horse cannot force "maybes" to either be or not be produced, since if it sends a signal it is always possible that it will be received between the "end" of one period and the "begin" of the next (and so will not be considered).

3.2 Examples of Half-Bit Channels

"Naturally occurring" half-bit channels are often caused by the sharing of finite resources. The buffering mechanism of the Gypsy specification language [Goo 86] is a typical example of a shared resource which is finite in the case of bounded buffers. A Gypsy buffer is a communication mechanism in which messages are received in the same order they are sent, but which may be received at an arbitrarily later time than they are sent. To receive a message, a Gypsy procedure explicitly receives from a named buffer; and if the buffer is not empty, the procedure will receive the message which is at the head of the queue. The head message is then removed from the queue. If the buffer is empty, the procedure "blocks", meaning that it is suspended until the buffer has something in it. To send a message, a procedure also names a buffer. If the buffer is not full (determined by its length bounds), then the message is put at the end of the queue. If the buffer is full, the procedure blocks until there is space on the buffer for the message. The nice thing about the buffering mechanism is that it is dependable: if a procedure tries to send to a buffer, the next thing it experiences will always be a successful send; and if a procedure receives from a buffer, the next thing it experiences will always be a successful receive. (However, in either case, it is possible the procedure will never experience anything; it may never become unblocked.)

For a Trojan horse and a spy to communicate, they must share some resource. In the case of buffers, there are two ways for them to share a buffer in keeping with Bell-LaPadula access rules:

1. The spy may send to a buffer which the Trojan horse is capable of reading from. Since the Trojan horse is presumably higher-level than the spy, this set-up is allowed by the rule "write up".
2. The spy and the Trojan horse may both send to a buffer that is read by some high-level or trusted procedure.

In the first case, a finite buffer may be used as a positive half-bit channel. To use the buffer in this way, the spy sends enough messages to the buffer to fill it up, and then sends one more. If the Trojan horse does nothing, then the last send will be unsuccessful, since the buffer will be full. If the Trojan horse wants to indicate a "yes" it only needs to receive from the buffer. When the spy successfully completes his send, he knows that the Trojan horse has said "yes". The Trojan horse cannot say "no", however, because if the send is unsuccessful, the spy's procedure will never become unblocked to inform the spy of the fact.

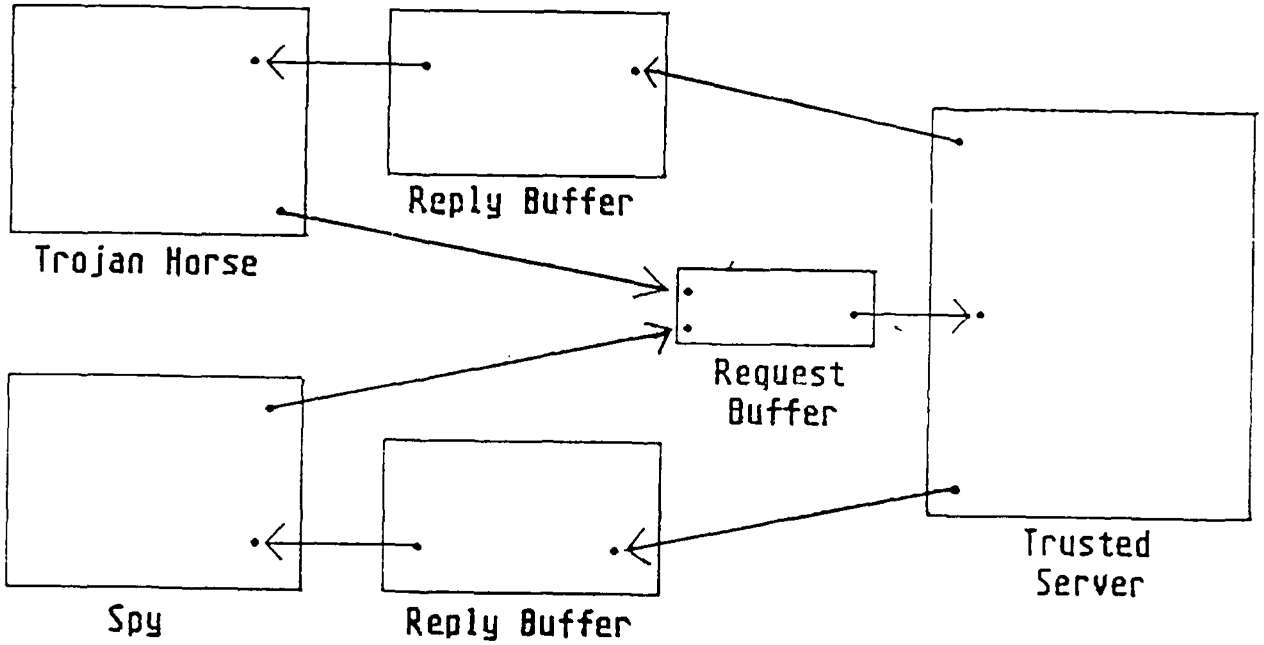

Figure 3.7: A Negative Half-Bit Channel from Bounded Buffers

In the second case, the finite buffer may be used as a negative half-bit channel. The mechanism for this is slightly more complicated. Suppose that the shared buffer is used to request some service to be performed by a trusted procedure. There are two buffers involved: the shared buffer for sending requests to the trusted procedure, and a private buffer for the spy to receive replies. We will assume that the trusted procedure receives requests one at a time, and sends the reply before taking another request. In this case, if the length of the request buffer is m and the length of the spy's reply buffer is n, the spy can send at most $n + m + 1$ messages before receiving a reply from the reply buffer. In the maximal case, the request buffer will be full, with m requests, the reply buffer will be full, with n replies, and one more request will have been taken from the request buffer; but the corresponding reply will not be sent (since there is no more room in the reply buffer). Now, the spy can receive from the reply buffer, making one more space. Then he can send one more request.

If this last send is successful, it will mean that the Trojan horse *did not* send a request. If it had, its request would have filled up the buffer, making it impossible for the spy to send another request. It is impossible for the trusted procedure to take any more requests from the buffer, since there is nowhere to put the replies[1].

The set-up creating a half-bit channel using bounded buffers is illustrated in figure 3.7.

[1] Actually, in this case, if the Trojan horse *does* send to the buffer, it will lock everything up hopelessly. An alternative strategy for the spy is for him to execute an "await" statement instead of a send. An await statement allows a procedure to either send to one buffer or receive from another, depending on which buffer is available. If an await is used, the spy will never be permanently blocked; the Trojan horse's message will cause the spy's procedure to choose to receive a reply instead of sending to the request buffer. If the spy manages to send to the request buffer, he knows that the Trojan horse *did not* send a request.

3.3 Converting a Negative Half-Bit to a Positive Half-Bit

If one feels that a negative half-bit channel is acceptable, but that a positive half-bit channel is not, then one must face the upsetting fact is that it is possible to convert the one into the other. All that is necessary is a simple converter process. This converter is illustrated in figure 3.8.

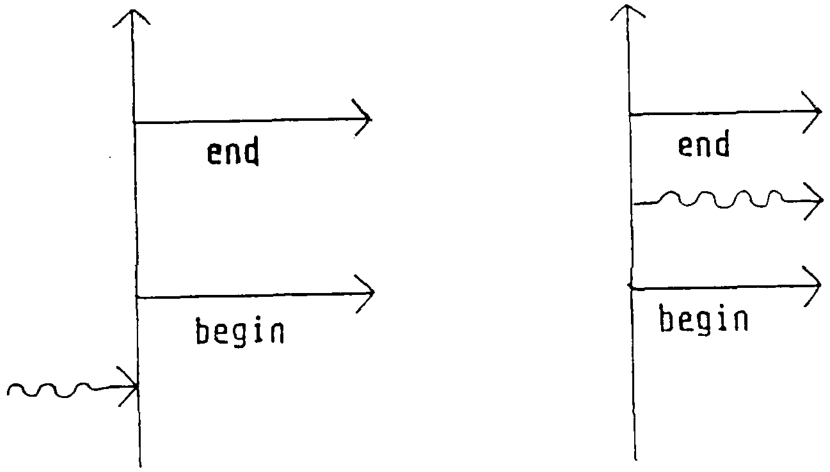

Figure 3.8: A Negative to Positive Channel Converter : Possible Histories

The converter process takes in classified inputs and sends out classified and unclassified signals. If it has not received a classified signal since the last unclassified output, then it outputs the sequence: "begin", classified signal, "end" ("begin" and "end" being two unclassified signals). If it has received a classified signal in the period since the last unclassified output, then it outputs the sequence "begin", "end" (skipping the classified signal).

Now, the effect of this converter is to produce a classified output if and only if it does *not* receive a classified input. If this is combined with the negative half-bit channel of figure 3.6, the resulting system is pictured in figure 3.9.

This combined system has the property that if a classified signal is received by the converter, then the half-bit system will output "no" or "maybe". If no classified signal is received, the half-bit system will output "maybe". Therefore, the combined system acts like a positive half-bit channel (with "no" to indicate a signal instead of "yes").

The converter described above is actually a secure process, by most reasonable definitions of security. To see this, one only needs to look at the unclassified outputs; they are always "begin" followed by "end". The sequence of unclassified outputs is completely unaffected by classified inputs, and so the process should be considered secure. The only objection one can make to the converter process in isolation is that it produces classified outputs even though there have been no classified inputs. Such outputs are "write-ups" which are disallowed on some systems for integrity purposes; this prevents unclassified users from being able to insert irrelevant material into classified files.

3.4 Two Half-Bits Make a Whole Bit

Is it possible to add two positive half-bit channels to make a one-bit channel? It sounds plausible, and in fact the answer is "yes". This is demonstrated in figure 3.10. In this composite system, we have systems $\mathcal{A}$ and $\mathcal{B}$, both of which are positive half-bit channel systems. $\mathcal{A}$ indicates that it has received a classified signal by outputting "yes" or "maybe"; and $\mathcal{B}$ indicates that it has received a classified signal by outputting "no" or "maybe". If either machine fails to receive any classified input between two unclassified requests, it will output "maybe". Now, to combine $\mathcal{A}$ and $\mathcal{B}$ into a full-bit channel, we need an additional server process $\mathcal{S}$. If $\mathcal{S}$ receives classified signal a it will send a classified signal to $\mathcal{A}$. If it receives classified signal b, it will send a classified

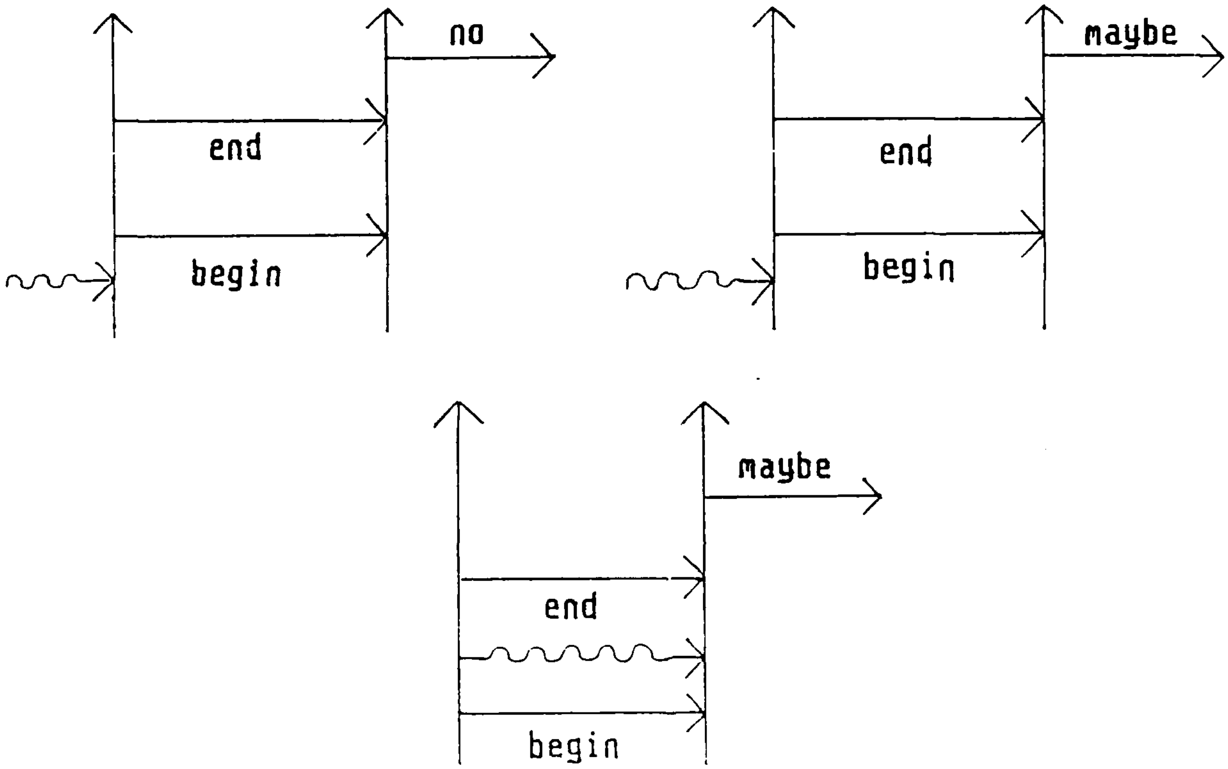

Figure 3.9: A Positive from a Negative : Histories

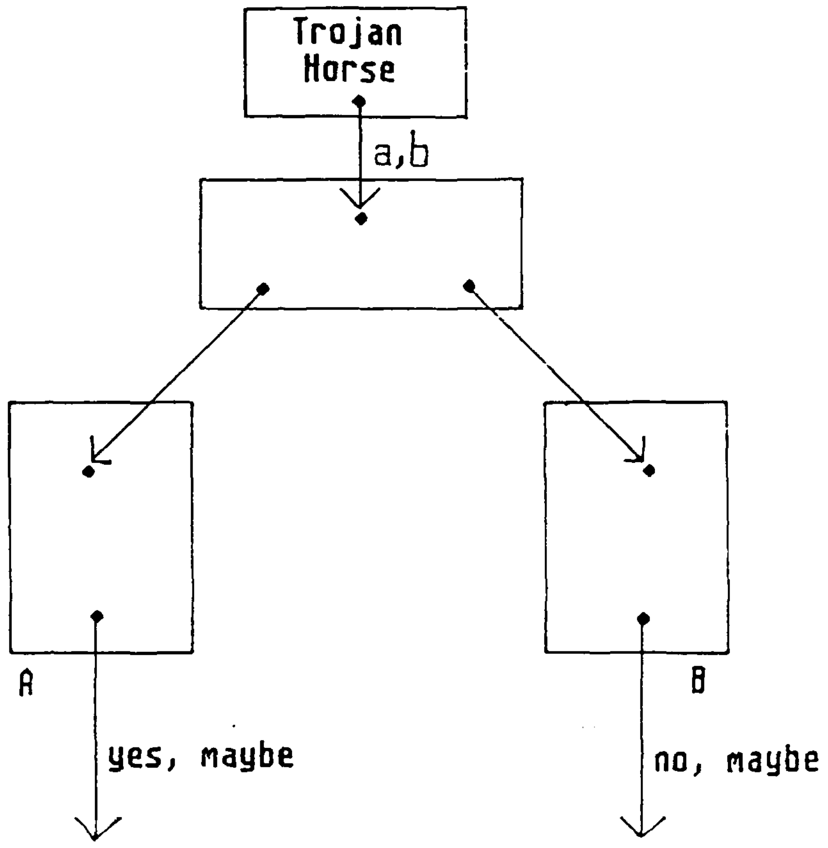

Figure 3.10: A Full-Bit Channel from Two Half-Bits

signal to $\mathcal{B}$. Therefore, if the combined system receives a classified a, it will output an unclassified "yes" or "maybe", and if it receives a classified b, it will output "no" or "maybe". The combined system is thus a noisy one-bit channel.

3.5 Avoiding Covert Channels

From the previous discussions it should be evident that if one wishes to avoid full-bit covert channels in systems, then he must make sure that at least one of the following holds:

- There are no Trojan horses capable of exploiting half-bit channels. This might be accomplished by disallowing "write-ups" if the only channels are negative half-bit ones.
- There must be a notion of a secure system and a corresponding notion of the legal "hook-up" of systems such that for any collection of secure systems that are connected in legal ways, the resulting composite system is secure. (At least in the sense that it does not allow any one-bit channels.)

Since it may be difficult to tell when a program is a Trojan horse program without detailed, costly analysis, it appears that the second route is the more practical. It makes security analysis more modular, in that to make a composite system secure, it is only necessary to make sure that each component is secure and that all the connections are legal. We turn next to the discussion of several security properties, and under what circumstances they are composable properties.

Chapter 4

Goguen–Meseguer Machines and MLS Noninterference

Goguen and Meseguer[Gog 84] take the principal notion behind security to be that of ***noninterference*** rather than information flow. Noninterference can be used to give information flow restrictions; rather than saying that person $\mathcal{A}$ is not allowed to receive information from source $\mathcal{B}$, one can instead say that source $\mathcal{B}$ is not allowed to ***interfere with*** person $\mathcal{A}$. The link between the two statements is the plausible assumption (which can be formalized and proved; this was done by Sutherland[Sut 86]) that $\mathcal{A}$ cannot learn anything about $\mathcal{B}$ unless $\mathcal{B}$ has some effect on (or interferes with) something visible to $\mathcal{A}$.

To make the notion of noninterference more precise, Goguen and Meseguer give an abstract model for a class of information processing systems and define noninterference for that class. Their original model was for state machines, and for general noninterference policies (and not simply for multilevel security). We will modify their treatment in two ways:

1. We will only consider MLS noninterference properties.
2. We will give the definition in terms of input and output sequences rather than state machines.

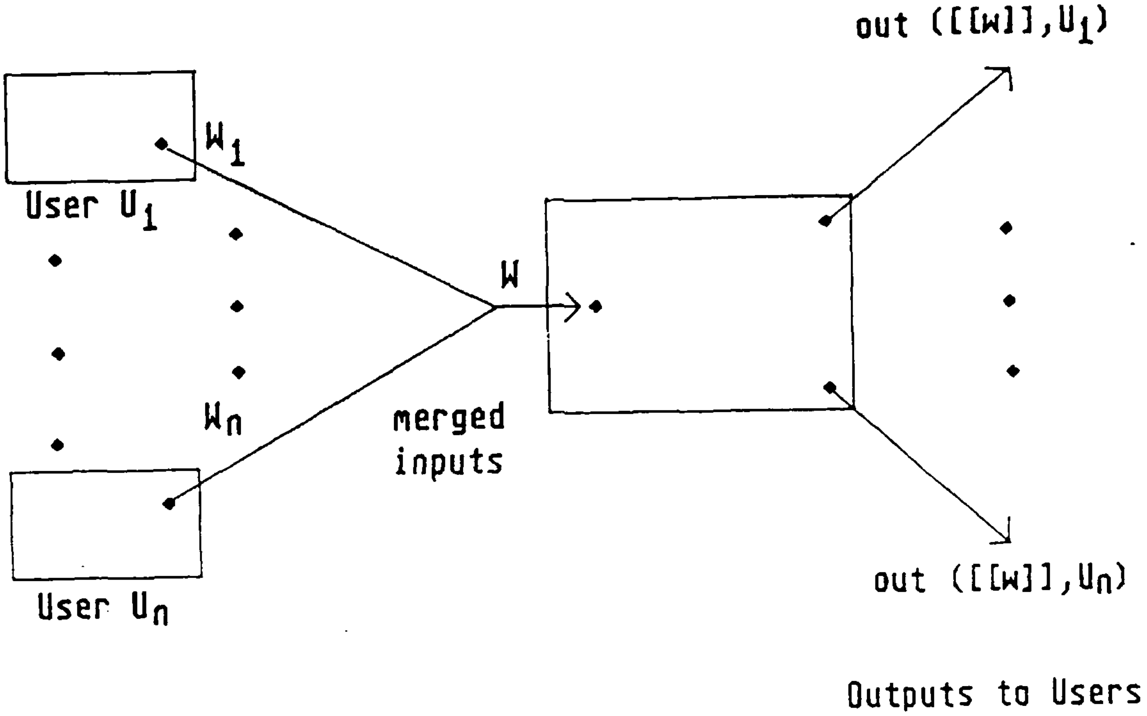

Figure 4.1: Goguen-Meseguer Machines

In their model, there is a collection of users, which we will call u_1, u_2, ... u_n. Each user u_i issues a sequence of commands w_i. The command sequences w_1, w_2, ... w_n are merged to form the sequence w. The system computes a function of its input sequence for each user, and the appropriate output is sent to each user. This setup is illustrated in figure 4.1.

Each user u_i has a security level assigned to him. If the system obeys the MLS noninterference policy, then for every pair of users u_i and u_j, if the level of u_i is not less than or equal to the level of u_j, then the inputs of u_i may not interfere with the outputs of user u_j.

Formally, letting $out([[w]], u_j)$ be the outputs to user u_j resulting from input sequence w, and letting $PG_{u_i}(w)$ be the result of purging from w all inputs from user u_i, the requirement of noninterference becomes, for all i, j, w:

$$level(u_i) \not\leq level(u_j) \rightarrow out([[w]], u_j) = out([[PG_{u_i}(w)]], u_j)$$

4.0.1 Assumptions Behind the Model

The model of information processing assumed by Goguen and Meseguer is not completely general. By assuming that the output is a function of the input sequence, they have restricted the set of systems to which their model applies in at least two ways:

1. They only consider deterministic systems. For nondeterministic systems, the output is not a function of the input sequence, since more than one output sequence can result from the same input sequence.

2. They only consider uninterruptable systems. This restriction is closely related to the first, because a system with interrupts will look nondeterministic when one looks at the observable behavior. For example: consider a system which allows the user to abort a computation that is taking too long. For such a system, the same input sequence, the start command followed by the abort command, can give rise to two

different output sequences depending on whether the abort command comes before or after the calculation is completed. (If it comes in time, the system may respond with "Ok". If it comes too late, the system may instead respond with "No processes running".) The fact that interrupts result in nondeterminism reflects the fact that timing of events is not considered in the model.

The above assumptions, while they rule out some perfectly reasonable systems from consideration, have the nice feature that if a system design is proved to be MLS noninterfering, and if the output function really captures everything about the system that is visible to the user, then any implementation will also be noninterfering, since *everything* visible to the user would already be decided at the design stage. If interrupts and nondeterminism were included, then there would be the possibility that the implementation could affect the precise way that the interrupts or the nondeterminism worked, thus possibly invalidating the proof of security.

4.0.2 How to Make a System Deterministic

When inputs are coming from a human being, it may be plausible to treat the system as *infinitely* fast; that is, the system can complete any calculation between two inputs from the user. However, when inputs come not from a human, but from another machine, this is no longer a reasonable assumption; the possibility arises that the system may be given inputs faster than they can be handled. Since we are ultimately going to be discussing the composition of several machines, it's necessary to give this matter a little thought. On first analysis, there seem to be three ways of handling the untimely arrival of inputs:

1. Assume that there is an unbounded input buffer. Then it would be possible for the system to leisurely take the inputs as it has time for them, and the result is the same as if the inputs had come more slowly. This solution has the disadvantage of being impossible to implement in this finite world. However, there may be situations in which a finite

but very long buffer can be treated as if it were infinite, if it is known that there is a very small probability of the buffer becoming full.

2. Assume that there is a finite buffer, and that the process or person making the inputs will "block" when the buffer is full; that is, patiently wait for the buffer to have space before making an input. This is the solution used by Gypsy, and as we have argued in Chapter 3, this kind of buffer leads to "half-bit" channels. It may be possible that such half-bit channels are not exploitable in systems obeying Goguen–Meseguer noninterference.

3. Assume that there is a finite buffer, and that if the buffer is full, the next message will be dropped (either with or without notifying the sender).

One alternative to blocking inputs when a buffer is filled is to respond with an error message "Busy" if a message arrives when the system is not ready to take it. As shown in figure 4.2, this leads to nondeterminism, exactly as the existence of interrupts did.

The Problem with Composition: Nondeterminism

Even if we assume infinite buffers, nondeterminism creeps in when we consider connecting several processes. This is illustrated in figure 4.3. Here, we see two processes f and g, each of which is deterministic. However, they are connected together in such a way that their outputs are merged. Two different output sequences of the resulting composite system are possible, depending on which process, f or g, finishes first. Thus the resulting system is nondeterministic (although determinism would be restored if the relative processing speeds of f and g were taken into account.)

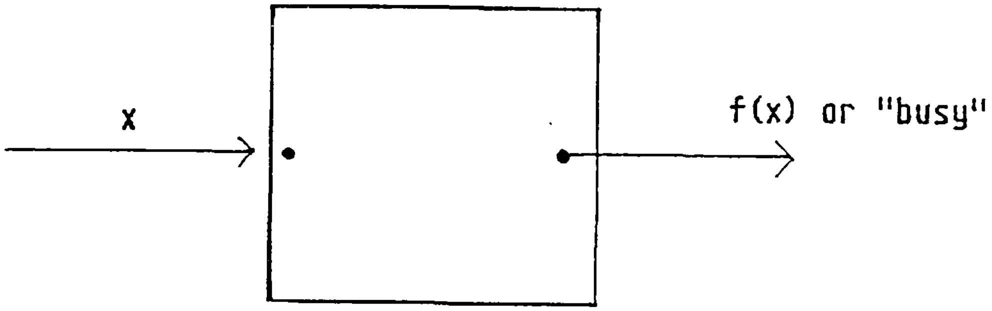

Figure 4.2: Not Blocking Inputs

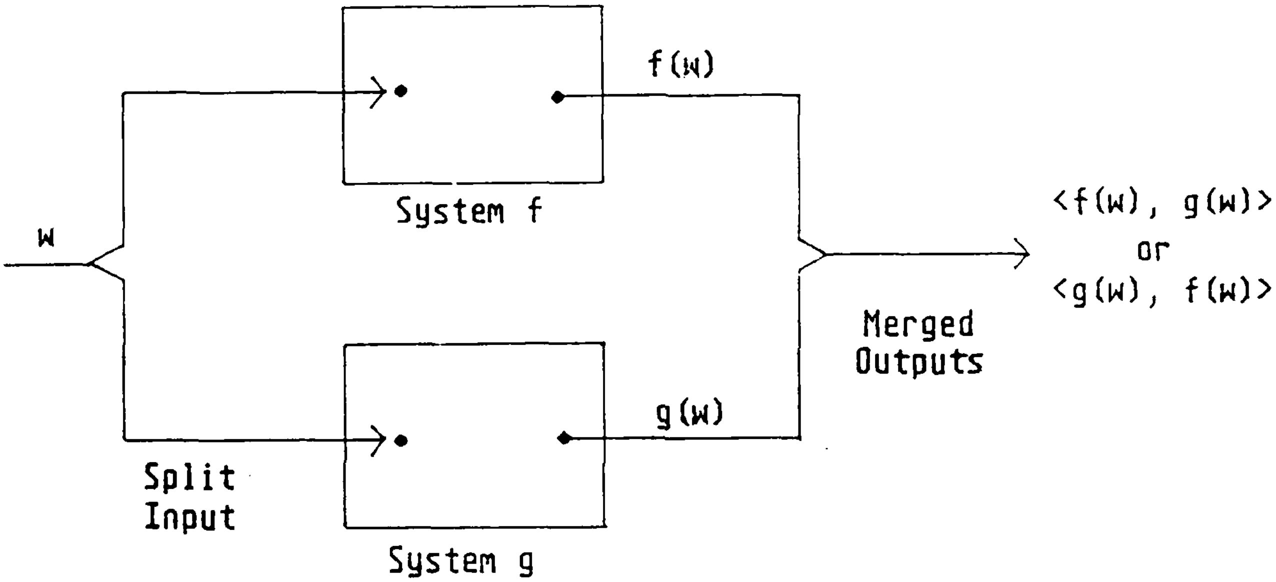

Figure 4.3: The Nondeterministic Composition of Deterministic Machines

4.0.3 How to Preserve Noninterference Under Composition

To preserve the deterministic MLS noninterference property when processes are composed it is necessary to require the following:

- Because bounded buffers seem either to lead to covert channels or to nondeterminism, all processes must communicate through unbounded buffers,

- There must be no merging of the outputs of different processes; that is, two different processes may not both send to the same process or buffer, since this leads to nondeterminism, which cannot be handled by the Goguen and Meseguer definition of noninterference. This rules out the kind of uses of multiprocessing in which the results of many parallel computations are funneled into a centralized source, where they can be acted upon as they arrive.

If these restrictions are bearable, then Goguen and Meseguer can be used in composing systems. If they seem hard to live with (or if the assumption of infinite buffers seems unrealistic), then there is a motivation to look for a generalization of Goguen and Meseguer noninterference. As we mentioned earlier, the assumption of deterministic processes, which seems to be at the root of the problems, is not apparently relevant to security. We turn now to a general formulation of information flow which does not make the assumption of determinism.

Chapter 5

A Deducibility Model of Information Flow

In this chapter, we give a brief overview of Sutherland's model of information flow. A more complete discussion can be found in [Sut 86].

5.1 Background Knowledge of Users

We consider the question of what it means for an observer (a flesh and blood human or an artificially intelligent machine or computer program) to obtain some desired information from an information processing system. Since it takes information to be able to process information, we first consider what kind of initial knowledge the observer may have about the system and his possible interactions with it.

First of all, we assume the user knows the design of the system. In principle, then, the observer knows the set H of all possible histories of the system. Depending on the details of his knowledge, a history may be, to name some examples, the sequence of states of the system, the sequence of signals exchanged with the environment (other observers), or the real-time behavior of the system, etc.

Second, we assume that the observer knows what his interface with the system is. This defines the relationship between his observations and the history of what has gone on in the rest of the system. This relationship can be expressed as a function, called the observer's *view*, from histories into observations. For example, his view may be the sequence of characters printed out on his computer screen.

Third, we assume that the observer knows what information he wants to obtain. This information can also be expressed as a function from histories into some value set. For example, he may wish to know the contents of some secret file at a particular time, which will in general be a function of the history of the system (especially the writes to the file) up until that time.

5.2 Inference

How much of the desired information can the observer discover through his observations? We can state this question more generally: The observer's observations in a history are given by some function on the history. For example, this function may take the history of all events that occur and keep only the events which are visible to the observer's terminal. Likewise, the information is another function of the history. We will call such functions which take a history and return information *information functions*. Thus the general problem of information flow is that given any two information functions f_1 and f_2, what can be deduced about the value of f_2 given the value of f_1?

Let us return to our observer, and assume his view is defined by the information function f_1, and that his desired information is defined by function f_2. Suppose that his observations correspond to f_1 having value v_1. He does not know what has gone on outside of his view, so he does not know which out of the set of possible histories is the actual one. However, he has a constraint on the possible histories; whatever history has taken place, it must be consistent with what he has seen. Thus the actual history must be in the set

$$H_1 \equiv \{h \in H \mid f_1(h) = v_1\}$$

This is the set H_1 of all histories h consistent with the observation that $f_1(h) = v_1$.

Now, for each history h in H_1, the observer can in principle calculate the value his desired information function, f_2 has in that history. This gives the set of possible values of f_2 consistent with the fact that f_1 has value v_1. These values are then given by the set

$$V_2 \equiv \{f_2(h) \mid h \in H_1\} \text{ which is equal to } \{f_2(h) \mid h \in H \;\&\; f_1(h) = v_1\}$$

5.3 Information Flow

The possibilities in V_2 determine whether the observer has learned a little or a lot about his desired information function; the smaller the set, the more he has learned. If the set V_2 is a singleton, then he has learned exactly the value of his desired information; there is only one possibility consistent with his observations. (If V_2 is empty, then he has made a mistake in his calculations somewhere along the line.) When should we say that he has learned nothing at all? Obviously if his observation has not helped him; if the set of possibilities for f_2 in the restricted set of histories H_1 is the same as the range of possibilities in the full set of histories H. This motivates the following definition of "no information flows from f_2 given that f_1 has value v_1":

$$No\_Flow(f_2; f_1, v_1) \equiv (\; \{f_2(h) \mid h \in H \;\&\; f_1(h) = v_1\} = \mathrm{range}(f_2) \;)$$

This is immediately seen to be equivalent to the following definition:

$$No\_Flow(f_2; f_1, v_1) \equiv \forall h_2 \in H \;\exists h \in H \;[f_2(h) = f_2(h_2) \;\&\; f_1(h) = v_1]$$

This definition can be taken to mean that *every* possible value of f_2 is consistent with the observation that f_1 has value v_1; nothing has been ruled out.

Now, what does it mean to say that it is *impossible* for an observer with view f_1 to ever obtain information about the value of f_2? It means that for all possible values of f_1 there is no flow from f_2. This is formalized as

$$\mathit{Never\_Flow}(f_2; f_1) \equiv \forall h_1 \in H[\mathit{No\_Flow}(f_2; f_1, f_1(h_1))]$$

This can be unravelled into the following definition:

$$\mathit{Never\_Flow}(f_2; f_1) \equiv \forall h_1, h_2 \in H \exists h \in H[f_2(h) = f_2(h_2) \ \& \ f_1(h) = f_1(h_1)]$$

This definition can be taken to say that there is never a flow from f_2 to f_1 if and only if the two functions are *independent*; that is, if they can be assigned values independently by the appropriate choice of the history h. This definition has one profound peculiarity: the *Never_Flow* relation is symmetric—if there is never a flow from f_1 to f_2, then there is never a flow from f_2 to f_1. This is a surprising result at first sight, since it is common to consider information flow as having a direction (and without a sense of directionality it is not clear that we have modelled information "flow" at all). Intuitively, information flows from a data file to a user when the file is read, and flows from a user to a data file when the file is written. However, in terms of the inference model developed above it seems that in both cases information flows in both directions; when a user writes a file, he knows something about the file afterwards—namely, what he just wrote into it. Going a step further, if someone knows the contents of a file, then he knows something about any other user who reads the file: namely what that user read. This perhaps unintuitive result will have consequences for the person wishing to use the information flow model in a definition of security. We will see in the chapter on event systems, chapter 9, that it is possible, with the proper interpretation of the information functions of the system, to reintroduce an intuitive notion of "flow".

5.4 Deducibility Security

The deducibility security for an information processing system can be defined by giving the following objects:

1. H, the set of possible histories
2. F, a set of information functions on H
3. $V \subseteq F$, the set of views
4. *hidden*, a function from V into F

The first three items are familiar from the above discussion. The fourth item, *hidden*, is a function which for each view gives what information should be hidden from that view. In the case of multi-level security, there will be a view for each user, and *hidden* might be the information which gives all information whose sensitivity is greater than or incomparable to the clearance of the user.

The deducibility security of the system is then defined as follows:

$$\forall f \in V[\mathit{Never\_Flow}(\mathit{hidden}(f); f)]$$

This definition says that information never flows into a view from the information that should be hidden from that view.

In using the deducibility model it is necessary to decide

1. How should the histories be described?
2. What are the view functions?
3. What is the "hidden" information for each view?
4. Is deducibility security an adequate requirement for the security of systems?

5. If not, then what additional requirement is needed?

In chapter 2 we argued that the answer to the fourth question is "no" if we are considering nondeterministic systems or systems that will be hooked up to larger systems. In the following chapters we will show how the general theory of deducibility security can be instantiated for systems described as processes, and how it can be strengthened to a hook-up security property which is more adequate for our needs.

Chapter 6

Process Specification

One of the most common techniques developed to rigorously analyze the behavior of programs is to view a program as an algorithm for computing functions of a data type. The advantages that lie in associating programs with mathematical functions are many, chief among them being implementation-independence and extensionality. To know how to use a program in conjunction with other programs one does not need to know how the program was constructed (often a messy tale of "while" loops and assignment statements); it is enough to know the extensional heart of the program: the function it computes.

In an information processing system with many jobs executing concurrently, the idea of a program as simply an algorithm becomes at best inconvenient and at worst false. The most important objects in such systems are not algorithms, but processes.

A process sometimes seems to compute a function. For example, a process such as an interpreter can be thought of as computing and outputting to the screen the value of the expressions input from the keyboard. However, a process fails to be functional in two respects.

Dependence of Outputs on the Process History

The first reason that a process does not simply compute a function of the inputs is that the output at a given time depends on the process history. In the case of the interpreter, for instance, the value of an expression may depend on the assignments made to variables many inputs ago. One way out of this difficulty is to introduce the notion of the current state of the process. In that case, the output at a given time is a function not of the last input, but of the state. Such an approach for specifying the behavior of systems is taken, for example, by S.D.C.'s Formal Development Methodology with the specification language Ina Jo [Loc 80].

Powerful techniques have been developed by Floyd [Flo 67], Hoare [Hoa 69], Gries [Gri 81], Dijkstra [Dij 76], and others to combine algorithmic and the state transformation notions of programs. The idea behind these techniques is to view a program as describing the action of a state machine in which the "states" are assignments of values to all the variables in the program. The state transformations can be specified by statements, called "Hoare triples", of the form: $\{P\}S\{Q\}$ which means:

> If predicate P holds of the state of the system initially, and program S runs to completion, predicate Q will hold afterwards.

For a program to "run to completion" it must run until it terminates normally; i.e., without raising any error conditions.

Another way to view the process is as a means of computing not a function of one input, but a function from the entire sequence of past inputs. In this second view, a process is a function from streams (or sequences) of data to streams of data. This change of emphasis, from single inputs to streams of inputs, allows one to analyze programs such as operating systems which never "halt" and so as algorithms only compute the totally undefined function. A Hoare-triple such as $\{P\}S\{Q\}$ is useless when S represents a program which is *intended* not to halt; the statement "If S halts, then Q is true afterwards" is a vacuous statement in such a case.

Non-Determinism

Real processes fail to be functional in a second way; the output is in general not completely determined by the input stream. Reasons for this nondeterminism include the lack of complete knowledge of the system and the presence of inherently unpredictable timing effects arising from the interaction of many entities: processes, devices, and users. This extra complication is manageable with the state machine approach of Ina Jo, or the Hoare-triple approach, since it only involves introducing nondeterminism into the state transition relations, those rules which describe which states may follow a given state. However, the reasons which make the extensional view of programs attractive also make state machines unattractive; a description of a process as a state machine is not independent of implementation. Two processes which "do the same thing" do not necessarily have the same state-machine description. Of course, there are ways around this deficiency; one could develop a formalism for defining what it means for two state machines to be observationally equivalent, and then every state machine description could be understood to represent an equivalence class of observationally equivalent processes. We intend rather to follow the approach of Hoare[Hoa 85]. There, a process is identified with its observable behavior from the start.

Processes As Event Systems

A process is an abstraction which is intended to capture the dynamic character of a component of an information processing system. Although there are some subtleties involved in choosing a sufficiently complete characterization of a process(see [Hoa 85]), we will simply use the input-output relation for the process.

A process is viewed as a "black box" into which one puts things and out of which one gets things in response. For our purposes, we may as well regard the "things" as messages or signals of some sort, but bear in mind that this interpretation is unnecessarily restrictive; one may reasonably regard, say, a Coke machine as a process with the input objects being coins and the output objects being bottles of Coke.

We assume that the complete specification of a process involves giving a complete set of events for the process and the set of its possible behaviors as described by traces of the process. Here we only consider the events corresponding to the input or output of some "thing", and a trace is defined to be a finite history of the process, giving the sequence of events in the order they occurred.

For our later discussion on the security of processes, it will be necessary to make a clear distinction between input events and output events. Intuitively, output events are under the control of the process, while input events are controlled by some entity, such as a person or another process, which is "external" to the process. To formally define the "cause" of an event is tricky, but for our purposes it is sufficient to assume that while the output sequence is constrained by the process' specification, the input sequence is completely unrestrained except for the type of the inputs. (In the Coke machine example, one may put in any coins in any order, but one cannot choose to ignore the type restrictions and input a credit card, or say, a banana. The size of the coin slot enforces a kind of type restriction which cannot be circumvented without damaging the operation of the machine.)

For the present discussion, we ignore the timing characteristics of the process. In this untimed model of processes, the behavior of the process is completely determined by the sequence of events involving that process. We assume that events can be considered as atomic, so that no more than one event may happen at a time. The time between events is ignored.

Note that by assuming that events are atomic, we are being restrictive to some extent about what can reasonably be interpreted as an event. It is not reasonable for realistic systems to assume that arbitrarily powerful operations, such as "transmit file", can be atomic; in general, such actions will be broken up into a sequence of smaller actions. In modeling a system at a high level of abstraction, the modeler must keep the atomicity of operations in mind; the reliability of the conclusions he reaches about the behavior of the system are contingent upon the extent to which the actual system behaves "as if" the complex actions are atomic and uninterruptable. It is necessary to demonstrate that the actual implementation faithfully simulates the high-level abstract view of the system.

Chapter 7

A Model for Systems

Our model for both processes and systems is the *event system.* Event systems are based on the processes of Milner [Mil 80] and Hoare [Hoa 85]. For the purposes of security, we only wish to consider two systems different if that difference is manifest in a difference of behavior; that is, if there is a difference in the *traces*, or possible sequences of events.

By basing our model of systems on the behavior of systems, we insure that any definition of security for that model will be ostensible, and so two systems which behave the same will be equally secure (or insecure). Also, as has been shown by Hoare and Milner, the process model of systems (of which our model is a special case) is very general, and can be applied at almost any level of abstraction, and so our definition of security based on this model is similarly level-independent.

In addition to the information needed to describe the behavior of the event system, we also need to give information about the security aspects of the system. We will call an event system together with the security rating of its events a *rated event system.*

7.1 Possible Histories

For our purposes, then, a rated event system $\mathcal{R}$ is completely specified by giving its event structure and its security structure. The event structure can be described by the four sets $\langle E, I, O, T \rangle$. E is a set of *events* which includes all signals exchanged between the system and the external world, as well as all externally visible transitions of the system. I and O are disjoint subsets of E, the input events and output events, respectively. T is the set of traces, which are finite sequences of events representing the possible histories of runs of the system.

Intuitively, inputs are events which are *externally caused*, and outputs are events which are *internally caused*. That is, the system has no control over what signals come in, and the external world likewise has no control over what comes out. This idea can be illustrated by considering an example of a personal computer as an event system. The inputs, in the simplest case, are keystrokes, which are caused by the user typing at the keyboard. The outputs are the characters displayed on the screen, which are caused by the system.

We formalize the notion that inputs are "uncaused" by the system by requiring that the set of traces T be closed under extension by an input—that is, an input can occur at any time, and the system can never prevent an input from occurring (but it may choose to ignore an input if it is not ready to handle it). To formalize the notion that outputs are "caused" by the system, we require that the outputs of a given system be inputs to any other system to which the outputs are visible. In the example of a personal computer, the outputs, which are the characters appearing on the screen, become "inputs" to the user if he is watching the screen.

The interpretation of traces, as we have said, is that a trace represents a history of the system up to some time, that history being described by giving the sequence of events which occurred during it. The model has only "soft-time"; that is, the relative ordering of events is given, but not the real-time delay between events.

The choice of considering only "soft-time" properties is made for the simplicity of the formalism. If one wishes to include real-time, it can be done approximately by using "timing events" as follows:

- A special process, the "clock process", is introduced which regularly outputs timing events, or "ticks", to every other process.
- Each other process has a characteristic time for each output, which is expressed as the number of input "ticks" that must be received before the output occurs.

The "ticks" are a way of simulating the passing of time. Such timing is necessarily discrete, but by choosing a "tick" to represent a very small interval of time (in other words, by requiring a very large number of "ticks" before any process can make an output), continuous time can be approximated to any degree of accuracy.

Using real time is not in general useful at the design level, since the real-time behavior of a system depends sometimes critically on implementation details, including hardware characteristics, which are not known at the design stage. However, a real-time analysis could be used on a completed implementation of the design.

7.2 Security Levels and Views

The security structure of a rated event system $\mathcal{R}$ which has the event structure $\langle E, I, O, T\rangle$ is given by the triple $\langle L, \leq, lvl\rangle$, where L is a set of security levels, $\leq$ is the security ordering on levels, and *lvl* is a function which assigns a security level to each event in E. For this rated event system, we can, for each level l, define the set of events, called the *view*, visible to that level: $vue(l) \equiv \{e \in E \mid lteq[lvl(e), l]\}$.

7.3 The Information Contents of Views

For every view, or set of events, v and every sequence of events τ, we define the restriction of τ to view v (denoted by $\tau \uparrow v$):

> $\tau \uparrow v$ is the subsequence of τ formed by discarding all events which are not in the set v.

For a set of sequences T', we will use $T' \uparrow v$ to mean the set resulting from taking the restriction to v of each sequence in T.

If a user is restricted to seeing only the events in view v, then $\tau \uparrow v$ is all the information available to that user during the history τ. The function which takes a history τ and returns the sequence $\tau \uparrow v$ we will call the *information function* of view v. Two histories which have the same restriction to view v will look the same to that view. Thus we can form the equivalence class of a trace τ generated by view v as follows:

$$\|\tau\|_v \equiv \{\tau' \in T \mid \tau' \uparrow v = \tau \uparrow v\}$$

The information that flows from the source view v_s to the target view v_t during history τ can be determined by the following considerations (as described in chapter 5) : If a user is not allowed to observe anything during history τ, then the set of possible sequences of events consistent with his information is of course just T. The set of possible values of the source information function is $T \uparrow v_s$. On the other hand, a user allowed to see view v_t will be able to deduce that the possible sequences of events consistent with his information is $\|\tau\|_{v_t}$, and the set of possible values of the source information function is $\|\tau\|_{v_t} \uparrow v_s$. If information has flowed from v_s to view v_t in history τ, then $\|\tau\|_{v_t} \uparrow v_s$ will contain fewer possibilities than $T \uparrow v_s$. (In the special case in which *all* the information in view v_s flows to v_t, $\|\tau\|_{v_t} \uparrow v_s$ will consist of only one possible value. The user with view v_t would then be able to deduce *exactly* what events took place in the source view v_s.) If the two sets of possibilities, $T \uparrow v_s$, and $\|\tau\|_{v_t} \uparrow v_s$, are the same, then no information will have flowed from v_s to view v_t. In this case the user with view v_t will have learned nothing through his observations that he could not have learned from his knowledge of the system alone.

7.4 Deducibility Security

Now we can define illegal information flow for the rated event system $\mathcal{R}$:

> In every trace τ in T, and for all levels l in L, no information shall flow from the set of inputs outside of the view, $I - vue(l)$, into the view $vue(l)$.

It can be shown that there is no illegal flow if and only if for every level l and every trace τ, there is a trace τ' in $\|\tau\|_{vue(l)}$ such that

$$\tau' \upharpoonright (I - vue(l)) = \langle\rangle$$

In other words, there is no illegal information flow for view $vue(l)$ during the history whose trace is τ if it is consistent for a user with that view to suppose that there were *no* inputs other than those visible to him. This form is often more convenient for proving that there is no illegal information flow[1].

7.5 Hook-Up Security

From the examples given in chapter 2, we see that information flow or deducibility security is quite fragile and context-dependent: systems which have

[1] A similar definition of security is given by Jacob[Jac 88]. His processes, however, are general CSP processes[Hoa 85], which do not distinguish between inputs and outputs. We believe that this distinction is crucial, since information only comes into the system through inputs.

no information flows when isolated can be completely insecure when connected with other systems. We therefore chose as our definition of security a property called "hook-up security" which we proved (chapter 9) implies that

- If a system is hook-up secure, then it has no illegal information flows.
- If two systems are hook-up secure, and they are "hooked up" so that some outputs of either system become inputs of the other at the same security level, then the resulting composite system is also hook-up secure.

Hook-up security therefore strengthens the information flow requirement in such a way that systems will not develop new illegal flows when hooked up with other secure systems. Hook-up security is a very strong noninterference requirement; it requires that inputs are able to interfere with, or affect, only subsequent higher-level outputs. (Goguen and Meseguer discuss a related noninterference requirement [Gog 84] for a more limited class of systems than we are considering. In particular, their requirement only applies to deterministic systems.) We turn now to the task of finding a suitable formal definition of hook-up security.

Chapter 8

Noninterference and Hook-Up Security

The informal statement of noninterference for the event system model that we are using is the following:

> *A system S has the noninterference property if for any level l and for any history of the system, the view at level l of that history is unaffected by inputs that are not visible to that view.*

What we would like to have is a property which is stronger than noninterference and is preserved by composition or hook-up. Thus the stronger property, which we will call "hook-up security", can be defined informally via a partially circular definition:

> *A system S_1 is hook-up secure if it has the noninterference property and for any system S_2 which is also hook-up secure, the composite system formed by hooking up S_1 and S_2 is hook-up secure.*

8.1 Why Care About Composability?

There are several reasons that a security engineer should care about having a definition of security that is composable:

- Few systems are entirely stand-alone; there are always incentives to hook small systems together into networks in order to share information or computational resources. In anticipation of future connections, the security engineer should make sure that whatever confidence he has in the security of his system carries over to the new composite system.
- Even for a single computer system, there are in general components which are partially independent, such as the disk drives, the CPU, terminals and printers. An accurate treatment of any system might require considering the interactions of several concurrent components.
- In the design of a large, complex system it may be easier to break up the system design into smaller subsystems and analyze the security of the components rather than try to prove the security of the system as a whole. For this "divide and conquer" approach to be successful, one needs a criterion for the security of the pieces which is sufficient to guarantee the security of the whole.
- Through time-slicing, concurrency is simulated on a single processor, and so a useful model for a system is that of several user processes and the operating system running "semi-concurrently". From the viewpoint of this model, it is not enough to consider the security of the operating system alone; it is necessary to consider the interactions of the operating system with the other programs. Protection against "Trojan horse" programs can thus be enhanced by considering the security of the operating system in the context of being hooked up to a collection of other, possibly malicious processes. (The idea connecting composability of security with protection against Trojan horses is due to John Millen of Mitre.)

Given that it is important to consider composability in the security of a system, the question becomes: what is a composable model of security?

8.2 Weak Noninterference

Now, in order to work with definitions of noninterference and hook-up security, we need to make them more precise by giving an unambiguous test for determining whether a system has the noninterference property or whether it is hook-up secure. In the following, we consider a series of precise definitions of noninterference, each stronger than the last, culminating in a definition of hook-up security. For each of the intermediate definitions, we demonstrate that the definition is not strong enough to be a definition of hook-up security by way of a counterexample; we exhibit two systems which individually meet the definition but whose parallel composition fails to meet the definition.

A system is said to have the *weak noninterference property* if for every level l and every trace τ there is another trace τ' having the same sequence of low-level inputs and outputs (inputs and outputs with levels less than or equal to l) as τ, but which has *no* high-level inputs (inputs with levels higer than or incomparable with l).

Intuitively, a system is noninterfering if high-level inputs do not interfere with low-level outputs. The definition of weak noninterference formalizes the following argument: The high-level inputs in a trace do not interfere with the low-level event sequence if the same event sequence is possible in a trace with *no* high-level inputs.

8.2.1 A Flaw in Weak Noninterference

To simplify the discussion, we will consider the case in which there are only two levels: *secret* and *unclassified*.

A problem with weak noninterference is that it is not strong enough to adequately capture the notion that changes in outputs are *caused* by earlier changes in lower-level inputs. We can see this by considering the following system which meets the first definition but which allows *secret* inputs to affect *unclassified* outputs.

Consider a system $\mathcal{A}$ which meets the first definition of noninterference, but in an especially devious way. System $\mathcal{A}$ has a single *unclassified* event: an output. It has a single *secret* input which it may receive on one of two possible communication ports, which we will call the *left* port and the *right* port, and also a single *secret* output. The *secret* output may occur at any time, and since the system is input-total, it can also receive a *secret* input at any time. An *unclassified* output may happen only if the earliest *secret* event that occurred previous to that unclassified output, if any, was an output. A legal trace of $\mathcal{A}$ is shown in figure 8.1.

Now, this system has the first noninterference property: given any trace, it is possible to modify it by adding or deleting *secret* inputs and to "fix up" the resulting sequence to a legal trace by modifying *secret* outputs. However, a peculiarity of this system is that the modification of the *secret* outputs sometimes needs to occur *before* the corresponding *secret* input. To see this, consider the trace in figure 8.1. If a *secret* input is added to the trace earlier than the *unclassified* output, then the result is the sequence pictured in figure 8.2, which is not a legal trace. This sequence can, however, be "fixed up" to get another legal trace by adding another *secret* output even earlier, as in the trace of figure 8.3.

Thus this system does not seem to enforce the normal sense of *causality*; adding a *secret* input seems to cause an output to appear in the past.

In figures 8.4, 8.5, and 8.6, we show corresponding systems of a second system $\mathcal{B}$, which we also assume has the first noninterference property.

We will assume that systems $\mathcal{A}$ and $\mathcal{B}$ have the same response to any *secret* input; if there is an *unclassified* output in a legal trace, then the earliest preceding *secret* event, if any, must have been an output. Described this way, it seems that *secret* inputs have some kind of effect on *unclassified* outputs: if a *secret* input comes early enough in a trace, it will prevent an *unclassified* output from occurring. However, since it is always possible to "fix up" a trace by inserting the appropriate *secret* outputs, the *secret* inputs do not *necessarily* interfere with the *unclassified* outputs.

Now, imagine connecting $\mathcal{A}$ and $\mathcal{B}$ in parallel. A legal trace of the combined

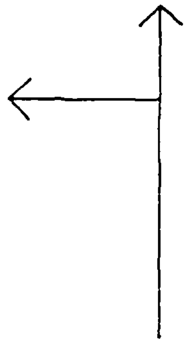

Figure 8.1: Original Legal Trace of System $\mathcal{A}$

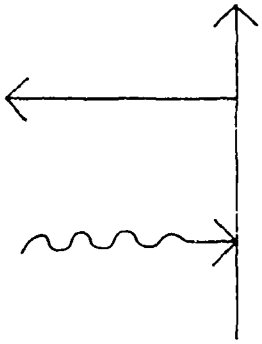

Figure 8.2: Add a *Secret* Input

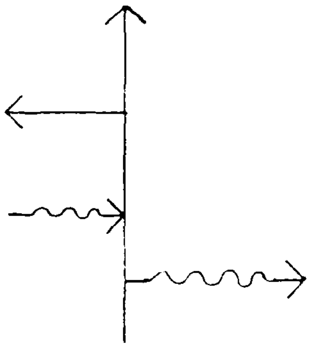

Figure 8.3: Another Legal Trace of System $\mathcal{A}$

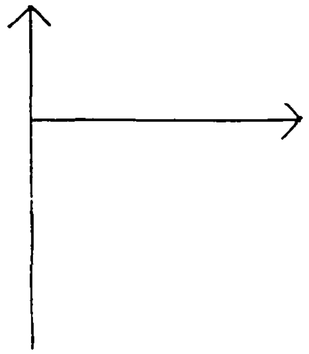

Figure 8.4: Original Legal Trace of System $\mathcal{B}$

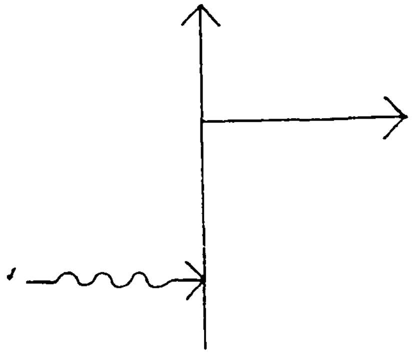

Figure 8.5: Add a *Secret* Input

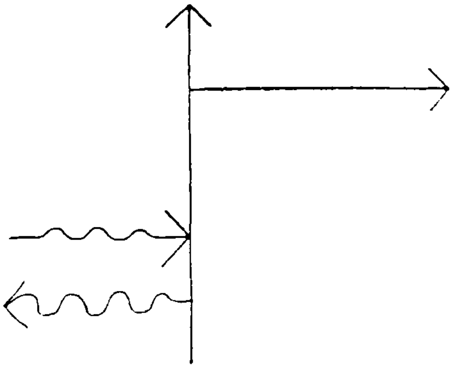

Figure 8.6: Another Legal Trace of System $\mathcal{B}$

system is shown in figure 8.7. In this figure, each system makes its *unclassified* outputs independent of the other, according to our convention that common events are shown between the two timelines.

In figure 8.8, we add a *secret* input to system $\mathcal{A}$. Thus we do not have a legal trace for the combined system; the sequence is legal as far as system $\mathcal{B}$ is concerned, but not as far as system $\mathcal{A}$ is concerned.

In figure 8.9, we "fix up" the trace for system $\mathcal{A}$ by allowing the output which corresponds to the input. By our convention, every output on the righthand side of the timeline of system $\mathcal{A}$ is an internal event, which is an input to system $\mathcal{B}$. Thus the sequence shown in figure 8.9 needs to be "fixed up" for $\mathcal{B}$. In figure 8.10, we "fix up" the sequence for $\mathcal{B}$ at the cost of introducing a new internal event, an input to system $\mathcal{A}$, which must be "fixed up" in turn.

It is obvious that there is no way to "fix up" the trace for both $\mathcal{A}$ and $\mathcal{B}$ simultaneously, so we must conclude that the addition of the *secret* input to system $\mathcal{A}$ in figure 8.8 must necessarily interfere with the *unclassified* outputs of the trace in figure 8.7. Therefore, the combined system does not have the first noninterference property even though (by *assumption*) the two component systems had the property. Thus, the first noninterference property cannot serve as a definition of hook-up security; hooking together two systems can destroy the noninterference.

8.3 Strong Noninterference

For a system that is weakly noninterfering, it is impossible at any time for a low-level user to deduce that a high-level input occurred earlier. However, it may be possible for a low-level user to deduce that an earlier high-level *output* has occurred. Such a deduction is not in general a security violation—because of write-ups a high-level output need not contain any high-level information. N^-vertheless, one could imagine strengthening weak noninterference so that deductions by a low-level user about inputs or outputs is forbidden. We may call this stronger property *strong noninterference.*

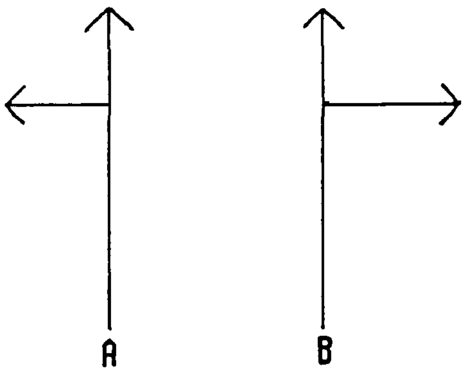

Figure 8.7: Original Legal Trace of the Combined System of $\mathcal{A}$ and $\mathcal{B}$

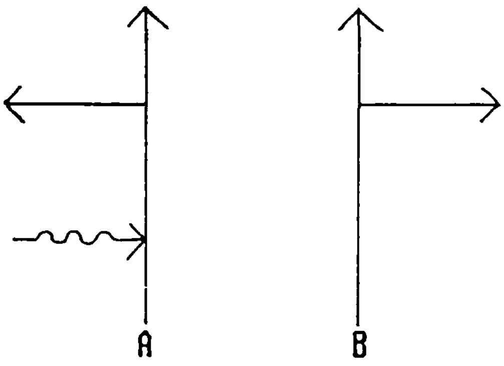

Figure 8.8: Add a *Secret* Input to System $\mathcal{A}$

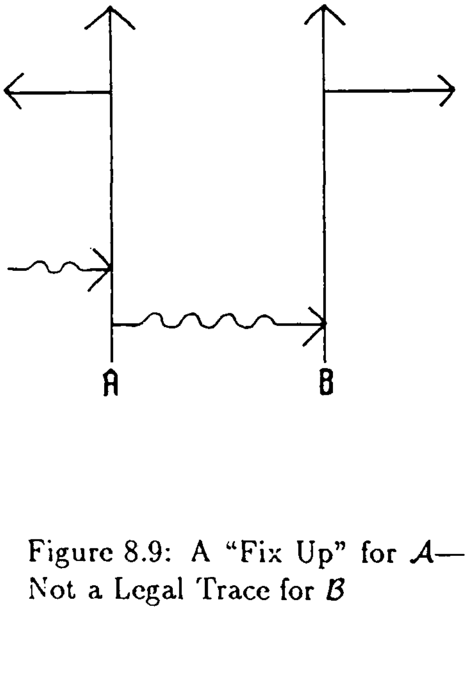

Figure 8.9: A "Fix Up" for $\mathcal{A}$— Not a Legal Trace for $\mathcal{B}$

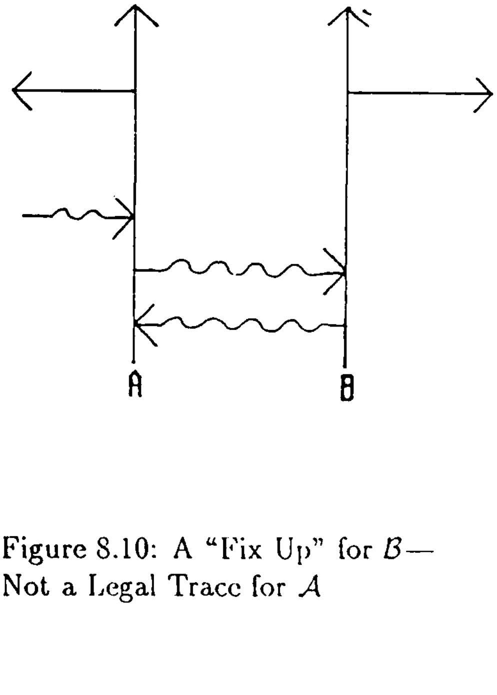

Figure 8.10: A "Fix Up" for $\mathcal{B}$— Not a Legal Trace for $\mathcal{A}$

Strong noninterference can easily be defined formally: For every possible trace τ of the system, the trace formed from τ by deleting all high-level events (inputs and outputs) is also a trace.

Doug Weber at Odyssey has proved that strong noninterference is a composable security property. (In the example in figure 8.10, it can be seen that a key feature of the systems that causes their composition to fail to be secure is that both of the systems allow unsolicited high-level outputs to occur.)

Despite the fact that strong noninterference is a composable property, it has several undesirable features:

- It forbids some systems that are obviously secure.

  To see this, consider a system which simply upgrades information; it repeatedly

  1. Takes in a low-level signal.
  2. Outputs the corresponding high-level signals.
  3. Outputs the low-level message "done".

  Now this system is manifestly secure; it can't possibly give away high-level information to low-level users, since it never even *receives* any high-level information to give away. The system fails to meet the requirements of strong noninterference, however, because for every trace containing the output "done", it is impossible to remove the high-level outputs and still have a legal trace. This shows that strong noninterference is an excessively strong requirement of systems.

- It is not preserved by upgrading outputs.

  Intuitively, upgrading outputs (increasing their security level) on a secure system should leave the system secure; there is *less* information available to low-level users than before. However, such an upgrade can change a system which obeys strong noninterference into one which does not, since the transformation may produce new unremovable high-level outputs.

- It permits systems which are intuitively insecure.

  Consider once again the system described in section 2.1.1. As you recall, it had the following characteristics:

  - There are two possible low-level input commands: *begin_eavesdrop* and *end_eavesdrop*.
  - If the low-level user issues *begin_eavesdrop*, and then at a later time issues *end_eavesdrop*, the system will respond by repeating to the low-level user the sequence of inputs made during the period between low-level commands, if any.
  - If no high-level inputs are made at all between *begin_eavesdrop* and the following *end_eavesdrop*, then the system will send to the low-level user a fake response made up of randomly selected outputs.

  This system obeys strong noninterference and is deducibility secure; a low-level user can never deduce with certainty that any high-level inputs at all occured. The system is nevertheless intuitively not secure, since there are circumstances in which the low-level user can be given high-level information.

For these reasons, we will reject strong noninterference as a candidate for a definition of hook-up security

8.4 Generalized Noninterference

We can try to correct the flaw in the first definition of noninterference by trying to make the relationship between inputs and outputs more causal—that is, we require that changing an input can only affect *later* outputs, and then only outputs of greater or equal level than that of the input changed. We thus modify our thought experiment for determining whether a system has the noninterference property:

- Take any legal trace τ_1 of a system $\mathcal{S}$.

- Break τ_1 into two pieces α and β_1 such that $\alpha^\wedge\beta_1 = \tau_1$. (The sequence $\alpha^\wedge\beta_1$ is the sequence α followed by the sequence β_1.)
- Modify τ_1 by inserting or deleting *secret* inputs in part β_1 to form a new final part β_2. The new sequence, τ_2, is equal to $\alpha^\wedge\beta_2$.
- $\mathcal{S}$ has the generalized noninterference property if and only if for any such τ_1 and τ_2, there is a legal trace τ_3 which differs from τ_2 only in *secret* outputs, and only in the *final part* of τ_2. The trace τ_3 is of the form $\alpha^\wedge\beta_3$ where β_3 differs from β_2 only in *secret* outputs.

For a system obeying this definition, if a legal trace of the system is modified by changing *secret* inputs, it should always be possible to "fix up" the resulting sequence to make it a legal trace by only changing *later secret* outputs. This definition is closer to our intuitions that an effect should follow its cause; then we can say that the effect of the *secret* input change is the resulting *secret* output change.

8.4.1 A Flaw in Generalized Noninterference

Generalized noninterference has one thing in its favor—according to it, the strange systems $\mathcal{A}$ and $\mathcal{B}$ described above do not obey generalized noninterference. We might then expect that any two systems meeting this stronger definition of noninterference will be such that hooking them up will preserve the noninterference. However, generalized noninterference, like weak noninterference, is not in fact strong enough to serve as a definition of hook-up security. This can be seen through the next example. Once again, we will call our component systems $\mathcal{A}$ and $\mathcal{B}$, but they will be improved models compared with the earlier systems with those names.

Let $\mathcal{A}$ be a system obeying generalized noninterference. System $\mathcal{A}$ has two *secret* inputs, one from a *left* channel and one from a *right* channel, and one *secret* output, which is sent out the *right* channel. It has three *unclassified* outputs: `stop_count`, `even`, and `odd`, with the first one being sent out the *left* channel, and the other two being sent out the *right* channel. System

$\mathcal{A}$ works as follows: the *secret* output and the event stop_count can occur at any time. The event even can only occur after stop_count, and only if the total number of *secret* inputs and outputs earlier than stop_count is *even.* Similarly, odd can occur only if the number of *secret* events earlier than stop_count is *odd.*

Figure 8.11 is a legal trace of system $\mathcal{A}$ with an even number of *secret* events. In figure 8.12 the sequence is modified by adding a *secret* input. This is no longer a legal trace, since there is now an odd number of *secret* events. In figure 8.13 the sequence is "fixed up" to a legal trace; a later *secret* output is added to bring the total number of *secret* events back to an even number.

It is clear that system $\mathcal{A}$ meets generalized noninterference: any modification of the *secret* inputs to a trace can be "fixed up" by adding or deleting the appropriate number of *secret* outputs, while leaving the *unclassified* events alone.

Let $\mathcal{B}$ be a second system with behavior almost identical to that of system $\mathcal{A}$, the differences being that

- For $\mathcal{B}$ stop_count is an input and not an output.
- The *secret* outputs of $\mathcal{B}$ are sent out the *left* channel of system $\mathcal{B}$.
- $\mathcal{B}$ can only receive inputs from its *left* input channel.

However, like system $\mathcal{A}$, *secret* outputs can occur at any time, and the *unclassified* events even and odd depend on the total number of *secret* events occurring before stop_count.

Figure 8.14 shows a legal trace of system $\mathcal{B}$ and figure 8.15 shows a modification of the trace by adding a *secret* input. Figure 8.16 shows a corresponding legal trace, in which an additional *secret* output occurs to compensate for the *secret* input. Since the effect of any *secret* input can be compensated for by a *secret* output, System $\mathcal{B}$, like system $\mathcal{A}$, has the generalized noninterference property.

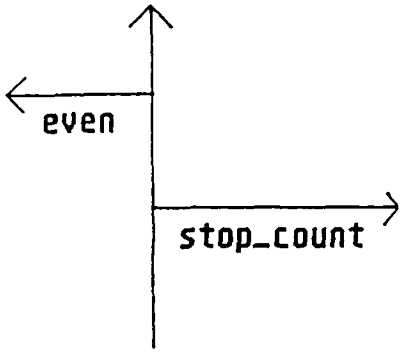

Figure 8.11: Original Legal Trace of Improved System $\mathcal{A}$

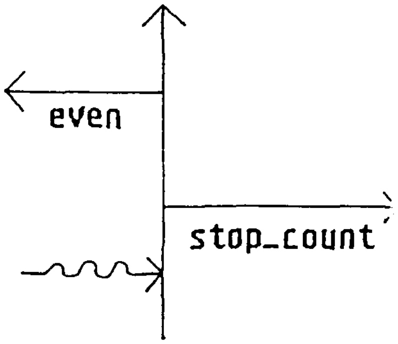

Figure 8.12: Add a *Secret* Input; Not a Legal Trace

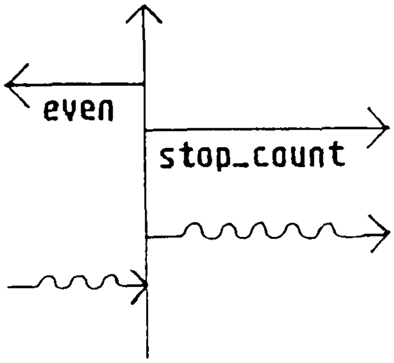

Figure 8.13: Another Legal Trace of $\mathcal{A}$; the *Secret* Input has a Response

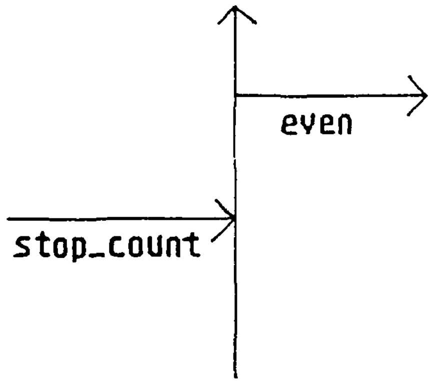

Figure 8.14: Original Legal Trace of System $\mathcal{B}$

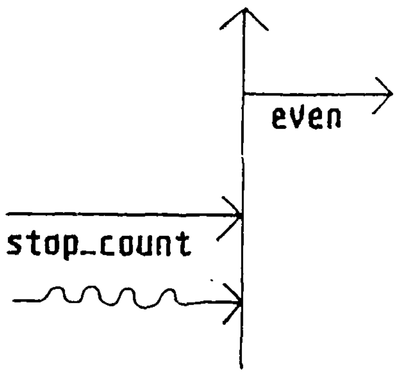

Figure 8.15: Add a *Secret* Input; Not a Legal Trace

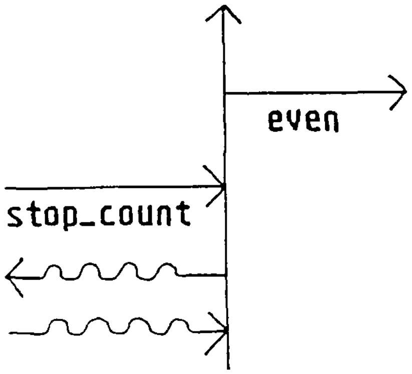

Figure 8.16: Another Legal Trace of $\mathcal{B}$; the *Secret* Input has a Response

Now, if we connect systems $\mathcal{A}$ and $\mathcal{B}$, then a possible legal trace of the combined system is shown in figure 8.17. Here there are an even number of *secret* events for each system: zero. In figure 8.18, we add a *secret* input to system $\mathcal{A}$. The result is no longer a legal trace, since there is now an odd number of *secret* events on system $\mathcal{A}$, although the unclassified output says **even**. In figure 8.19, we allow $\mathcal{A}$ to respond by making a *secret* output. This output happens to be an internal event of the combined system, and so is an input to system $\mathcal{B}$. This sequence is still not a legal trace, since now $\mathcal{B}$ has an odd number of *secret* events. In figure 8.20, we allow system $\mathcal{B}$ to respond by sending out a *secret* signal which is an input to system $\mathcal{A}$, to which $\mathcal{A}$ must respond in order to have a legal trace.

It is clear that it is impossible to simultaneously "fix up" the trace for both $\mathcal{A}$ and $\mathcal{B}$ and still have both systems output **even**; there must always be exactly one leftover *secret* input for either system $\mathcal{A}$ or system $\mathcal{B}$. For the combined system the addition of the *secret* input in figure 8.18 necessarily interferes with the *unclassified* outputs of figure 8.17: it is impossible to "fix up" the sequence in figure 8.18 to make a legal trace without changing those *unclassified* outputs. Thus the combined system does not obey generalized noninterference, even though the component systems do. Generalized noninterference thus cannot serve as a definition of hook-up security.

8.5 The Definition That Works: Hook-Up Security

The problem with generalized noninterference is apparent in figure 8.16: to "fix up" the trace of system $\mathcal{B}$ it is necessary to have a *secret* output occur *before* the next *unclassified* input. However, system $\mathcal{B}$ is no more able to control inputs than it is able to change the past. Therefore, in some cases, the "fixing up" is beyond the control of system $\mathcal{B}$ and so it never gets done. It is thus possible for an *unclassified* input to come when the system is not ready for it.

We therefore propose one more noninterference property, which we call "hook-

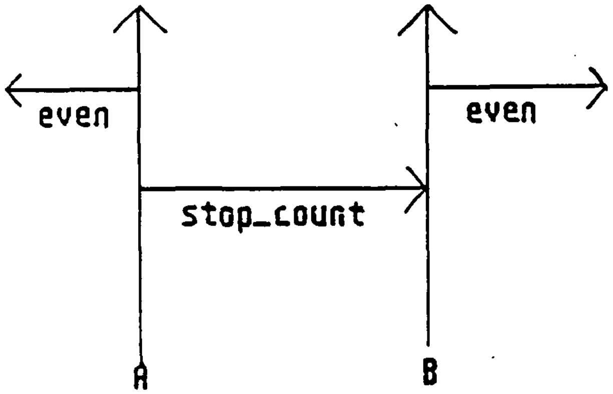

Figure 8.17: A Legal Trace of the Combined System

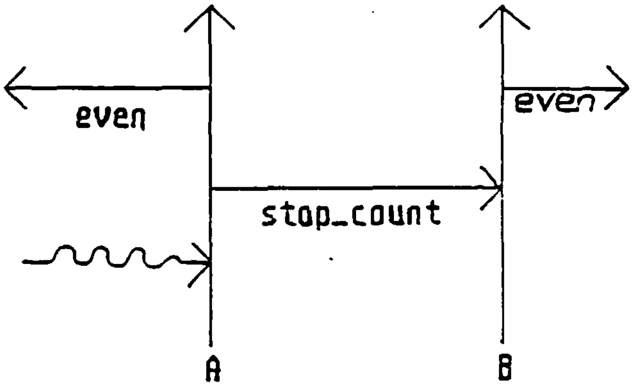

Figure 8.18: Add a *Secret* Input to $\mathcal{A}$; Not a Legal Trace for $\mathcal{A}$

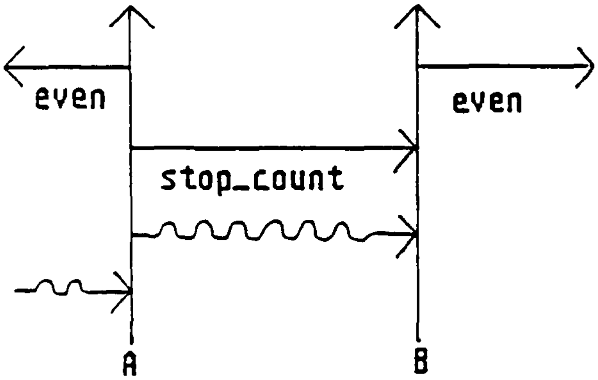

Figure 8.19: A "Fix Up" for $\mathcal{A}$; Not a Legal Trace for $\mathcal{B}$

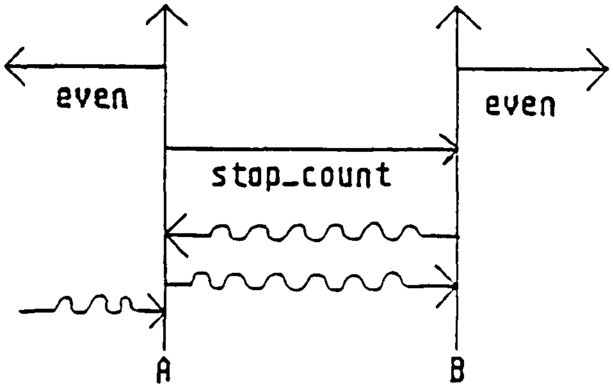

Figure 8.20: A "Fix Up" for $\mathcal{B}$; Not a Legal Trace for $\mathcal{A}$

up security" in expectation that it continues to hold when two systems are combined, or hooked up, to form a composite system. To be hook-up secure, the system must be able to "fix up" a trace following a change of *secret* inputs ***even if there is no time to do so before*** unclassified ***inputs occur.*** This means that regardless of the *secret* processing that needs to be done, the system must respond to any *unclassified* input in the same way.

When is a System Hook-up Secure?

- Take any legal trace τ_1 of the system.
- Break it into three segments α, β_1, and γ_1 such that $\alpha^\wedge\beta_1{}^\wedge\gamma_1 = \tau_1$, and such that β_1 is a sequence of inputs.
- Modify τ_1 by inserting or deleting *secret* inputs in β_1 and γ_1 to form sequence $\tau_2 = \alpha^\wedge\beta_2{}^\wedge\gamma_2$.
- S is hook-up secure if and only if for any such τ_1 and τ_2, there is a legal trace τ_3 of the form $\alpha^\wedge\beta_2{}^\wedge\gamma_3$, where γ_3 differs from γ_2 only in *secret* outputs.

  (Note: actually it is sufficient to only consider sequences such that γ_1 and γ_2 have *no secret* inputs. This is the form used in chapter 9. It is also sufficient to consider sequences β_1 and β_2 with at most one *unclassified* input and at most one *secret* input.)

This definition is a further generalization of noninterference—instead of saying that no change of *single secret* inputs may affect the future *unclassified* behavior of the system, we say that no change of the *secret* portion of any *sequence* of inputs may affect later *unclassified* behavior.

Looking once again at figure 8.16, we see that since the *secret* response to the *secret* input necessarily happens *before* the *unclassified* input, system $\mathcal{B}$ *is not* hook-up secure; if it were, then it would have been possible to make the first changed *secret* output occur after all the inputs immediately following the *secret* input. If this had been the case, then there would have been another

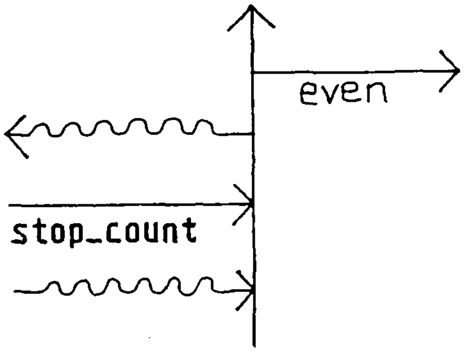

Figure 8.21: A Legal Trace for a Hook-Up Secure System *B*—The *Secret* Output Comes *After* the *Unclassified* Input

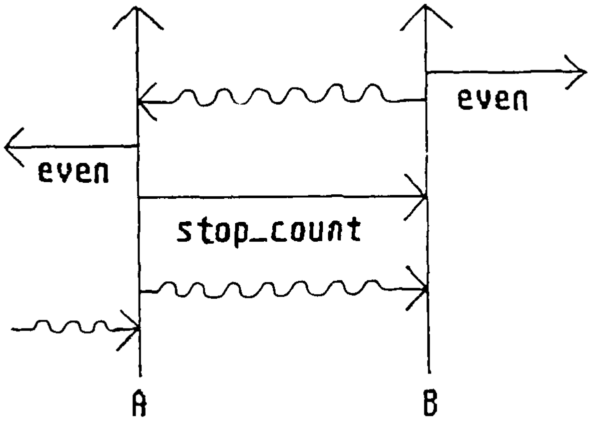

Figure 8.22: A Legal Trace for the Combined Hook-Up Secure System

legal trace of $\mathcal{B}$ as shown in figure 8.21, and correspondingly, there would have been a legal trace of the combined system as shown in figure 8.22. (Note: it is possible for two hook-up secure systems $\mathcal{A}$ and $\mathcal{B}$ to be combined into a system in which a *secret* input initiates a nonterminating exchange of *secret* signals between $\mathcal{A}$ and $\mathcal{B}$. If the component systems are hook-up secure then this infinite exchange cannot interfere with the *unclassified* processing. The unclassified processing must be done in the gaps between *secret* processing.)

It seems that the hook-up security fixes the flaws in the earlier noninterference definitions; it is plausible that two hook-up secure systems when hooked up in parallel form a hook-up secure composite system. (It is not necessary to stop at two systems; any number of systems can be hooked together by hooking up the first two to form a composite system, which is then hooked up to the third, and so on.) To be sure that hook-up security works this way, we need to prove it, as we do in chapter 9.

8.6 The Security Model and Its Properties

As stated in chapter 1, the requirements we had for a definition of security included:

1. Ostensibility—Security should be a property of the *behavior* of systems, and not of the way that behavior is implemented.

2. Description-Level-Independence—The security property should be usable at many different levels of detail of system descriptions.

3. Context-Independence—If a component is shown to be secure, it should be usable in any context requiring a secure component. In other words, it should be possible to be able to hook up secure components in a variety of ways without destroying security.

The first two requirements were addressed by using the event-system model described in chapter 7. We can only have confidence that the last requirement is addressed by proving that our formal definition of hook-up security has the hook-up property. We turn to the next chapter for a formal definition of hook-up security and the proof that it works as a context-independent definition of security.

Chapter 9

Event Systems and Hook-Up Security

In this chapter, we give formal definitions for the event systems and rated event systems which were discussed informally in chapter 7. We define the property called ***hook-up security*** and show that for deterministic systems, hook-up security implies Goguen and Meseguer's noninterference property for multi-level systems. We show that two systems which are hook-up secure and which assign the same level to those events common to the two systems can be hooked up to form a composite system which is hook-up secure.

To define the notion of hook-up security, we first define the notion of a ***restrictive*** view of a system. Intuitively, a view of a system is restrictive if no information leaks into that view from sources outside the view. Having defined a restrictive view, we define a system to be *hook-up secure* if the view associated with each security level (determined by the set of events that can be observed by users of that level or less) is restrictive. This captures the idea that no information should flow into the view associated with one level from the views associated with higher levels. The hook-up theorem is proved for the more general notion of restrictive views, and so is not limited to multi-level security.

9.1 Preliminary Definitions and Notation

Notation for Sequences

- $\langle\rangle \equiv$ the empty sequence
- $\langle e\rangle \equiv$ the sequence consisting of a single occurrence of event e
- $\alpha^\wedge\beta \equiv$ the concatenation of sequences α and β
- $\alpha \sqsubseteq \beta$ denotes that α is an initial segment of β, or alternatively, β *extends* α. In other words, for some δ, $\alpha^\wedge\delta = \beta$.
- $\alpha \sqsubset \beta$ denotes that α is an initial segment of β and that α is not equal to β; alternatively, β is a *proper extension* of α.
- $\alpha \sqsupseteq \beta \Leftrightarrow \beta \sqsubseteq \alpha$
- $\alpha \sqsupset \beta \Leftrightarrow \beta \sqsubset \alpha$

Notation for Sets

- $A^* \equiv$ the set of all finite sequences of events in A
- $A \cup B \equiv$ the union of A and B
- $A \cap B \equiv$ the intersection of A and B
- $A - B \equiv$ the set of all elements of A which are not elements of B

Definition 1 For all $\alpha \in E^*$ and $E' \subseteq E$, we will denote the *restriction of α to set E'* by $\alpha \upharpoonleft E'$. We define this recursively as follows:

$$\begin{aligned} \langle\rangle \upharpoonleft E' &\equiv \langle\rangle \\ \beta^\wedge\langle e\rangle \upharpoonleft E' &\equiv (\beta \upharpoonleft E')^\wedge\langle e\rangle \qquad \textit{provided } e \in E' \\ \beta^\wedge\langle e\rangle \upharpoonleft E' &\equiv \beta \upharpoonleft E' \qquad \textit{otherwise} \end{aligned}$$

9.2 Definition of an Event System

Definition 2 An *event system* ES is a structure $\langle E, I, O, T\rangle$, where E is a set of *events*, I and O are disjoint subsets of E, the *input events* and *output events*, respectively, and T is a set of *traces*, the legal finite sequences of events in E.

Let ES, ES_1, and ES_2 be the event systems $\langle E, I, O, T\rangle$, $\langle E_1, I_1, O_1, T_1\rangle$, and $\langle E_2, I_2, O_2, T_2\rangle$, respectively in the following.

9.3 The Simple Hook-Up of Event Systems

If one takes two systems running in parallel and allows them to communicate by exchanging events, then this composite system can be said to be a *parallel composition* or *hook-up* of the component systems.

Definition 3 We will say that ES is a *simple hook-up* of ES_1 and ES_2 if the following conditions hold:

$$\begin{aligned}
E &= E_1 \cup E_2 \\
I &= (I_1 - O_2) \cup (I_2 - O_1) \\
O &= (O_1 - I_2) \cup (O_2 - I_1) \\
I_1 \cap I_2 &= \{\} \\
O_1 \cap O_2 &= \{\} \\
(E_1 - I_1 - O_1) \cap E_2 &= \{\} \\
(E_2 - I_2 - O_2) \cap E_1 &= \{\} \\
T &= \{\alpha \in E^* \mid \alpha \uparrow E_1 \in T_1 \ \& \ \alpha \uparrow E_2 \in T_2\}
\end{aligned}$$

9.4 Multi-Level Security for Event Systems

Definition 4 A *security structure* SS for an event system $ES = \langle E, I, O, T\rangle$ is a structure $\langle L, \leq, lvl \rangle$, where L is a set of *security levels*, $\leq$ is a partial ordering on L, and lvl is a function from E to L, assigning a security level to each event.

Definition 5 A *rated event system* RES is a structure $\langle E, I, O, T, L, \leq, lvl \rangle$, where $ES = \langle E, I, O, T\rangle$ is an event system, and $SS = \langle L, \leq, lvl \rangle$ is a security structure for ES.

Let $SS = \langle L, \leq, lvl \rangle$, $SS_1 = \langle L_1, \leq_1, lvl_1 \rangle$, and $SS_2 = \langle L_2, \leq_2, lvl_2 \rangle$ be security structures for ES, ES_1, and ES_2, respectively in the following.

9.5 The Hook-Up of Two Rated Event Systems

Definition 6 Let RES be the rated event system $\langle E, I, O, T, L, \leq, lvl\rangle$, RES_1 be $\langle E_1, I_1, O_1, T_1, L_1, \leq_1, lvl_1 \rangle$, and RES_2 be $\langle E_2, I_2, O_2, T_2, L_2, \leq_2, lvl_2 \rangle$. We will say that RES is the *simple rated hook-up* of RES_1 and RES_2 if

- ES is the simple hook-up of ES_1 and ES_2.
- L_1 is a subset of L.
- L_2 is a subset of L.
- $\leq_1$ is the restriction of the relation $\leq$ to L_1.
- $\leq_2$ is the restriction of the relation $\leq$ to L_2.

- lvl_1 is the restriction of the function lvl to E_1.

- lvl_2 is the restriction of the function lvl to E_2.

9.6 Restrictive Views

Definition 7 A set of traces T will be called *input total* if it is closed under extension by inputs. That is:

$$\forall \alpha \in T, \forall x \in I, (\alpha^\wedge \langle x \rangle \in T)$$

If a system is input-total, then any input can always come at any time. Therefore, for such systems, no information is given to the sender of a signal by the fact that the signal is accepted, since it is always accepted, and so information only flows from the sender to the receiver.

Definition 8 A set of trac$\cdot$; T will be called *event-separable* if they are closed under taking initial segments. That is, if

$$\forall \alpha \in E^*, \forall e \in E, (\alpha^\wedge \langle e \rangle \in T \Rightarrow \alpha \in T)$$

The intuitive idea behind this definition is that a trace is supposed to represent the input-output history of the system up to some moment in time. Therefore, if $\alpha^\wedge \langle e \rangle$ is a trace, and the event e can be separated in time from the events in α, then there must have been a moment before e took place and after all the events in α took place. Thus α is a possible history.

In the following, we will consider only event systems with input-total and event-separable sets of traces.

Definition 9 We will use the word *view* to mean any subset of E.

This definition is motivated by the fact that any subset of events v defines an information function, or view of the set of traces:

$$\mathcal{F}_v(\tau) \equiv \tau \uparrow v$$

Two traces appear the same to view v if they differ only by events which are not in v. We will refer to events that are not in a view v as *hidden events* for that view.

Views for Multi-Level Security For a rated event system, the important views are those associated with the set of all events less than or equal to a given level:

Definition 10 Let *vue* be a function which takes a level and returns a subset of E such that:

$$vue(l) \equiv \{e \in E \mid lvl(e) \leq l\}$$

For each level l, $vue(l)$ is the set of events which can be legally seen by users of level l.

Definition 11 A set of traces T will be called *deterministic* if for all $\alpha \in T$, there is at most one $e \in E - I$ such that $\alpha^\wedge\langle e\rangle \in T$. That is, at any time, there is at most one output or internal event that may come next.

This definition contrasts with Hoare's definition of a deterministic process [Hoa 85].

9.6.1 Restrictiveness

We would like a composable noninterference property on views which corresponds to the hook-up security defined in chapter 8. We will define a seemingly weaker property, called *restrictiveness*, such that:

> If a view v of a system is restrictive, then for any legal trace, the last consecutive sequence of inputs may be modified by changing only inputs not in the view, and there exists another legal trace which differs from the modified trace in later outputs outside the view.

This operation is pictured in figures 9.1, 9.2, 9.3, and 9.4.

Definition 12 A view v is said to be *restrictive* if for all $\alpha \in T$, for all $\beta_1, \beta_2 \in I^*$, and for all $\gamma_1 \in E^*$, if

$$\begin{aligned} \alpha^\wedge\beta_1{}^\wedge\gamma_1 &\in T \\ \beta_1 \uparrow v &= \beta_2 \uparrow v \\ \gamma_1 \uparrow (I - v) &= \langle\rangle \end{aligned}$$

then for some $\gamma_2 \in E^*$,

$$\begin{aligned} \alpha^\wedge\beta_2{}^\wedge\gamma_2 &\in T \\ \gamma_2 \uparrow v &= \gamma_1 \uparrow v \\ \gamma_2 \uparrow (I - v) &= \langle\rangle \end{aligned}$$

9.6.2 A Fact About Restrictive Views

If a view is restrictive, then at any moment in any possible trace it is impossible for future inputs hidden from that view to interfere with the future sequence of events visible to that view. As shown in figures 9.5 and 9.6, this fact makes it possible to "strip" away all the hidden inputs from a trace starting at one point and to leave the visible events unaffected. In this figure, hidden events for the view are depicted with wavy lines, and visible events are depic'ed with straight lines.

This property will be important in the proof of the hook-up theorem of the next section. In the remainder of this section, we give a formal statement of this property and its proof.

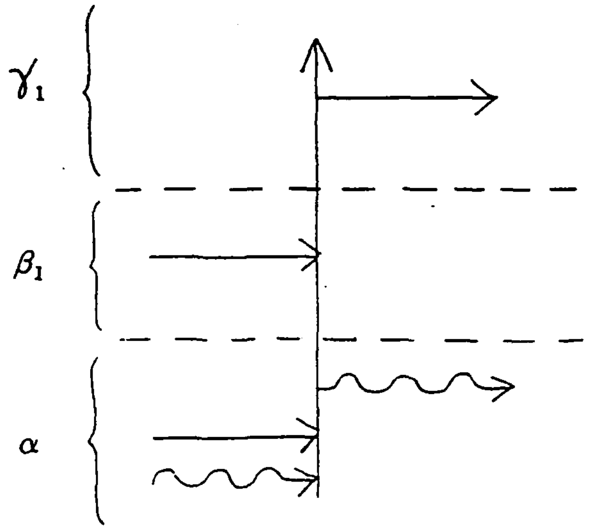

Figure 9.1: Original Legal Trace

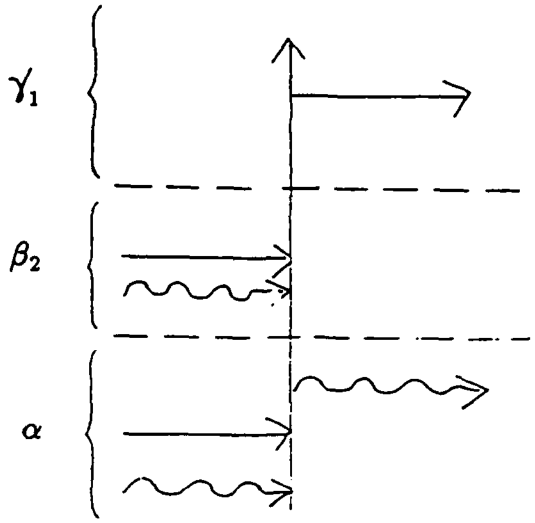

Figure 9.2: The Last Sequence of Inputs is Modified (Keeping the Inputs in View v the Same)

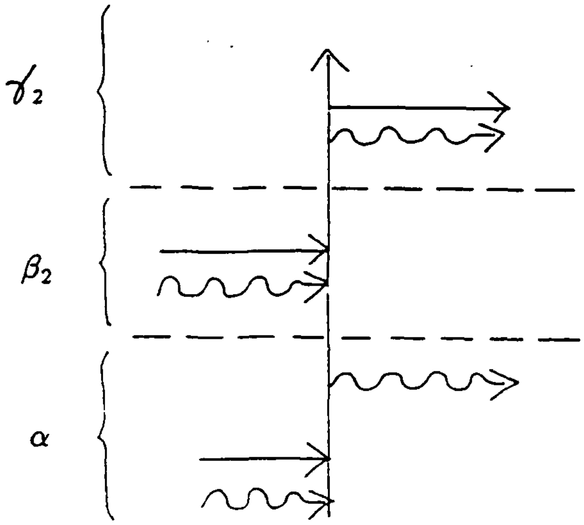

Figure 9.3: Another Legal Trace Must Exist with Only Outputs Not in v Modified

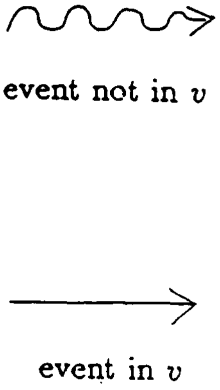

Figure 9.4: Legend

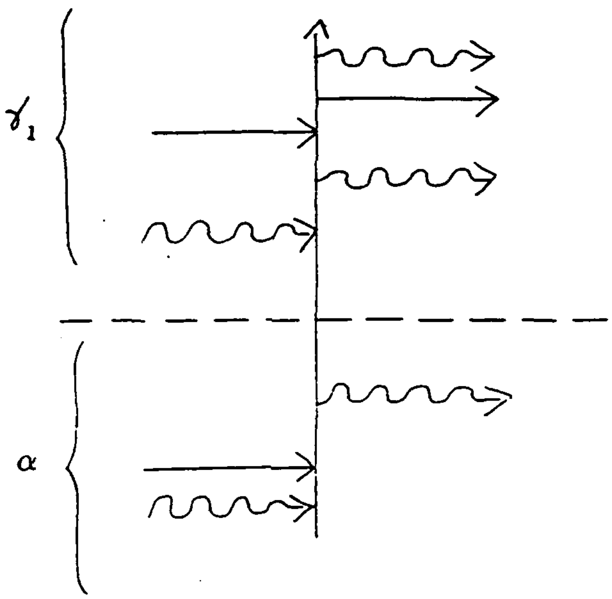

Figure 9.5: Original Trace

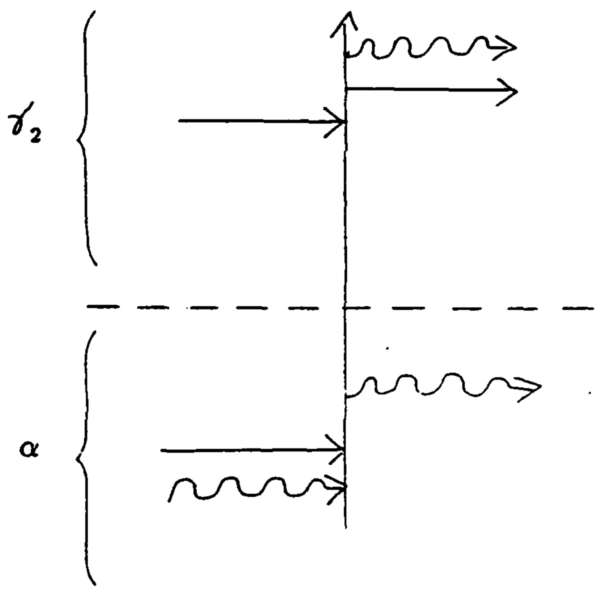

Figure 9.6: New Trace with Hidden Inputs Stripped Away

Theorem 1 *Let v be a restrictive view of event system* ES $= \langle E, I, O, T \rangle$, *and let α and γ_1 be elements of E^* such that $\alpha \hat{} \gamma_1 \in T$. Then for some $\gamma_2 \in E^*$, the following holds:*

$$\begin{aligned} \alpha \hat{} \gamma_2 &\in T \\ \gamma_2 \uparrow v &= \gamma_1 \uparrow v \\ \gamma_2 \uparrow (I - v) &= \langle\rangle \end{aligned}$$

Proof:

We prove the theorem by induction on the length of $\gamma_1 \uparrow (I - v)$.

Base Case: $\gamma_1 \uparrow (I - v) = \langle\rangle$

In this case, we can let γ_2 be γ_1.

Inductive Hypothesis: For all $\gamma \in E^*$, if the length of $\gamma \upharpoonright (I - v)$ is less than or equal to n and $\alpha^\wedge\gamma$ is in T, then there exists a $\gamma' \in E^*$ such that $\gamma' \upharpoonright v = \gamma \upharpoonright v$ and $\gamma' \upharpoonright (I - v) = \langle\rangle$.

Inductive Step: If the length of $\gamma_1 \upharpoonright (I - v)$ is $n+1$, then we can write γ_1 as $\delta_1{}^\wedge\langle e\rangle^\wedge\delta_2$, where $\delta_1 \upharpoonright (I - v)$ is of length n, and e is an element of $I - v$, and $\delta_2 \upharpoonright (I - v)$ is equal to $\langle\rangle$. Then we have:

$$\begin{array}{rcl} \alpha^\wedge\delta_1{}^\wedge\langle e\rangle^\wedge\delta_2 & \in & T \\ \langle e\rangle & \in & I^* \\ \langle\rangle & \in & I^* \\ \langle e\rangle \upharpoonright v & = & \langle\rangle \upharpoonright v \\ \delta_2 \upharpoonright (I - v) & = & \langle\rangle \end{array}$$

Since v is restrictive, we can use the definition of restrictiveness, definition 12, with the following instantiations:

- $\beta_1 \mapsto \langle e\rangle$
- $\alpha \mapsto \alpha^\wedge\delta_1$
- $\beta_2 \mapsto \langle\rangle$
- $\gamma_1 \mapsto \delta_2$

Thus there exists a sequence $\gamma_2' \in E^*$ such that

$$\begin{array}{rcl} \alpha^\wedge\delta_1{}^\wedge\langle\rangle^\wedge\gamma_2' & \in & T \\ \gamma_2' \upharpoonright v & = & \delta_2 \upharpoonright v \\ \gamma_2' \upharpoonright (I - v) & = & \langle\rangle \end{array}$$

Now, the length of $\delta_1{}^\wedge\gamma_2' \upharpoonright (I - v)$ is n, so we may use the inductive hypothesis. Thus, there exists a $\gamma_2 \in E^*$ such that

$$\begin{aligned} \alpha^\wedge\gamma_2 &\in T \\ \gamma_2 \uparrow v &= \delta_1{}^\wedge\gamma_2' \uparrow v \text{ which is equal to } \gamma_1 \uparrow v \\ \gamma_2 \uparrow (I - v) &= \langle\rangle \end{aligned}$$

This completes the proof. □

Corollary 1 *Let v be a restrictive view, and let α_1 be an element of T. Then, for some $\alpha_2 \in T$,*

$$\begin{aligned} \alpha_2 \uparrow v &= \alpha_1 \uparrow v \\ \alpha_2 \uparrow (I - v) &= \langle\rangle \end{aligned}$$

This is a special case of theorem 1, in which $\alpha = \langle\rangle$, and $\gamma_1 = \alpha_1$.

9.6.3 A Strong Form of Restrictiveness

In the definition of restrictiveness, there was the requirement that only the *last consecutive* sequence of inputs be modified. It is easy to remove this limitation, and thus come up with a seemingly stronger property:

Definition 13 A view v is said to be *strongly restrictive* if for all $\alpha \in T$, for all $\beta_1, \beta_2 \in I^*$, and for all $\gamma_1, \gamma_2 \in E^*$, if

$$\begin{aligned} \alpha^\wedge\beta_1{}^\wedge\gamma_1 &\in T \\ \beta_1 \uparrow v &= \beta_2 \uparrow v \\ \gamma_1 \uparrow v &= \gamma_2 \uparrow v \end{aligned}$$

then for some $\gamma_2' \in E^*$,

$$\begin{aligned} \alpha^\wedge\beta_2{}^\wedge\gamma_2' &\in T \\ \gamma_2' \uparrow (I \cup v) &= \gamma_2 \uparrow (I \cup v) \end{aligned}$$

With this stronger property, if a view v is restrictive, then it is possible to modify a legal trace in *any* manner, as long as the events in the view are left unchanged, and there will be another legal trace which differs from the modified trace only in outputs *outside* the view. Although this strong restrictiveness seems stronger than restrictiveness, it is easy to prove that the two properties are equivalent. This proof will not be given here in detail, but informally, if a view is restrictive, then it is possible to modify the inputs in a legal trace by the following procedure:

1. First, using the results of section 9.6.2, one can remove *all* the inputs outside of the view, starting at an arbitrary point in the trace.

2. Second, a different sequence of inputs outside the view can be added back, starting at the first point of modification and working one's way out. Because the later inputs outside the view have been stripped away, it is always possible to use the weaker version of restrictiveness.

Because the two forms of restrictiveness are equivalent, we will use the weaker form, since it is easier to work with formally.

9.7 The Hook-Up Theorem

In this section we prove that for a view of a two-component system to be restrictive, it is sufficient that the corresponding views for each of the component systems be restrictive.

We will assume that v is a restrictive view of event system ES, and that ES is the simple hook-up of ES_1 and ES_2. We will also assume that v is a view of ES such that $v_1 \equiv v \cap E_1$ is a restrictive view of ES_1, and $v_2 \equiv v \cap E_2$ is a restrictive view of ES_2. An important property of the sets v_1 and v_2 is that $v_1 \cap E_2 = v_2 \cap E_1$.

9.7.1 Merging Component Histories

Let *OK_SO_FAR* be a subset of $E^* \times E_1^* \times E_2^* \times v^*$ defined as follows:

$$\begin{aligned} \mathit{OK\_SO\_FAR} \equiv \{\langle \alpha, \gamma_1, \gamma_2, \gamma \rangle \mid\ & \gamma_1 \uparrow (I_1 - v_1) = \langle\rangle \ \& \\ & (\alpha \uparrow E_1)^\wedge \gamma_1 \in T_1 \ \& \\ & \gamma_1 \uparrow v_1 = \gamma \uparrow E_1 \ \& \\ & \gamma_2 \uparrow (I_2 - v_2) = \langle\rangle \ \& \\ & (\alpha \uparrow E_2)^\wedge \gamma_2 \in T_2 \ \& \\ & \gamma_2 \uparrow v_2 = \gamma \uparrow E_2 \} \end{aligned}$$

We can imagine trying to form a trace of the composite system as follows: First we split the system apart and form traces for each component system, and then we try to merge the two traces of the component systems to get a trace of the composite system. In this section we will show that if we have histories for each component system which agree on some initial segment α and agree on the common events in view v for the rest of the history, then it is possible to merge the two histories into a single history for the composite system with the same common initial segment and the same sequence of events in view v thereafter. This operation is illustrated in figures 9.7 and 9.8.

The progress made towards a composite trace can be coded by an element $\langle \alpha, \gamma_1, \gamma_2, \gamma \rangle$ of *OK_SO_FAR*: α is the trace of the composite system that we have created so far, γ_1 denotes the events that we still have to account for on the first system, γ_2 denotes the events we still have to account for on the second system, and γ is a possible extension of α for the composite system which gives constraints on the way events from the two systems are to be merged. If the constraints are satisfiable, then we are okay so far, and if we can make progress towards making α longer, then eventually we will have our composite trace. The constraints are requirements that the composite trace must look a particular way to view v. The next few theorems show that progress can be made.

Theorem 2 *Let* $\langle \alpha_1, \gamma_1, \gamma_2, \langle e \rangle^\wedge \gamma \rangle$ *be an element of* OK_SO_FAR. *Then there exist* $\alpha_2 \in E^*$, $\gamma_{1,2} \in E_1^*$, *and* $\gamma_{2,2} \in E_2^*$ *such that*

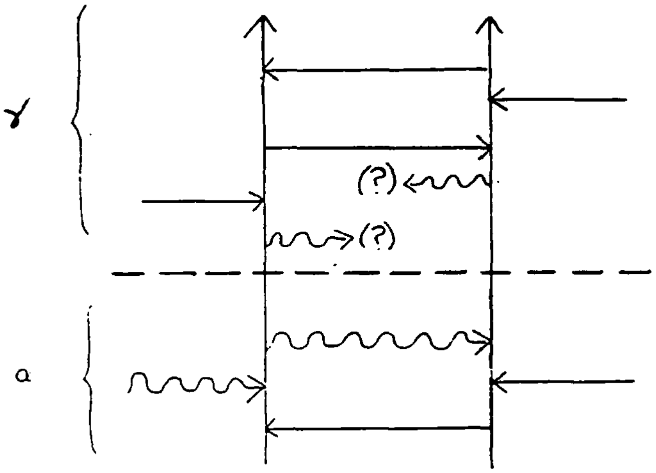

Figure 9.7: Partially Merged Histories

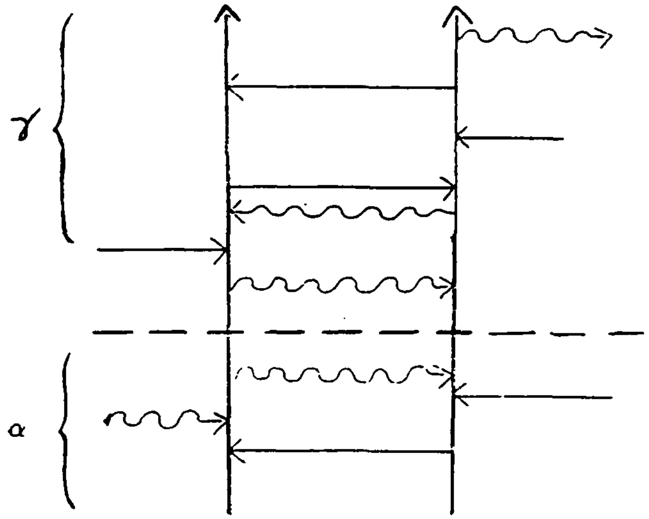

Figure 9.8: Merged Histories; Events in v Left Unchanged

$$\begin{aligned}\langle \alpha_1{}^\wedge\alpha_2, \gamma_{1,2}, \gamma_{2,2}, \gamma\rangle &\in \mathrm{OK\_SO\_FAR}\\ \alpha_2 \uparrow v &= \langle e\rangle\\ \alpha_2 \uparrow (I-v) &= \langle\rangle\end{aligned}$$

This theorem says that it is possible to take events from the fourth component of an element of *OK_SO_FAR* and move them to the first component. Since the fourth component tells us what we still have left to accomplish, and the first component tells us what we have already accomplished, this theorem says that progress is possible.

Proof:

Event e must be in v. Therefore, by the definition of simple hook-up, e must either be an event of one component system exclusively, or it must be an event which is an input for one system and an output for the other. The proof then divides into the following cases:

$$e \in v \cap (E_1 - E_2) \quad (9.1)$$
$$e \in v \cap O_1 \cap I_2 \quad (9.2)$$
$$e \in v \cap (E_2 - E_1) \quad (9.3)$$
$$e \in v \cap O_2 \cap I_1 \quad (9.4)$$

Cases 3 and 4 are obviously similar to cases 1 and 2, so it is sufficient to consider only the first two cases.

Case 1 $e \in v \cap (E_1 - E_2)$

By the definition of *OK_SO_FAR*, $\gamma_1 \uparrow v_1 = (\langle e\rangle\hat{}\gamma) \uparrow E_1$, so we can write γ_1 as $\gamma_{1,1}\hat{}\langle e\rangle\hat{}\gamma_{1,2}$, where $\gamma_{1,1} \uparrow v_1 = \langle\rangle$ and $\gamma_{1,2} \uparrow v_1 = \gamma \uparrow E_1$. Since, by definition of *OK_SO_FAR*, $\gamma_1 \uparrow (I_1 - v_1) = \langle\rangle$, then $\gamma_{1,1} \uparrow (I_1 - v_1) = \gamma_{1,1} \uparrow I_1 = \langle\rangle$.

Therefore, $\gamma_{1,1}$ consists of outputs and internal events for system ES_1. Since all events common to systems ES_1 and ES_2 must be inputs for one system and outputs for the other, we can conclude that $\gamma_{1,1}\hat{}\langle e\rangle \uparrow E_2$ consists of inputs to system ES_2.

Now, we have the following facts:

- $\gamma_2 \uparrow (I_2 - v_2) = \langle\rangle$
- $(\alpha_1 \uparrow E_2)\hat{}\langle\rangle\hat{}\gamma_2 \in T_2$
- $\langle\rangle \in I_2^*$
- $(\gamma_{1,1}\hat{}\langle e\rangle) \uparrow E_2 \in I_2^*$
- $((\gamma_{1,1}\hat{}\langle e\rangle) \uparrow E_2) \uparrow v_2 = \langle\rangle$ which is the same as $\langle\rangle \uparrow v_2$

Therefore, since v_2 is a restrictive view for system ES_2, we may use definition 12 with the instantiations:

- $\alpha \mapsto \alpha_1 \upharpoonright E_2$
- $\beta_1 \mapsto \langle\rangle$
- $\beta_2 \mapsto (\gamma_{1_1}{}^\wedge\langle e\rangle) \upharpoonright E_2$
- $\gamma_1 \mapsto \gamma_2$

Therefore, there is a $\gamma_{2,2} \in E_2^*$ such that

$$\begin{array}{rcl} (\alpha_1 \upharpoonright E_2)^\wedge((\gamma_{1,1}{}^\wedge\langle e\rangle) \upharpoonright E_2)^\wedge\gamma_{2,2} & \in & T_2 \\ \gamma_{2,2} \upharpoonright v_2 & = & \gamma_2 \upharpoonright v_2 \\ \gamma_{2,2} \upharpoonright (I_2 - v_2) & = & \langle\rangle \end{array}$$

By definition of *OK_SO_FAR*,

$$\langle(\alpha_1{}^\wedge\gamma_{1,1}{}^\wedge\langle e\rangle), \gamma_{1,2}, \gamma_{2,2}, \gamma\rangle$$

is an element of *OK_SO_FAR*. Also, we have

$$\begin{array}{rcl} \gamma_{1,1}{}^\wedge\langle e\rangle \upharpoonright v & = & \langle e\rangle \\ \gamma_{1,1}{}^\wedge\langle e\rangle \upharpoonright (I - v) & = & \langle\rangle \end{array}$$

as claimed. (The last two points follow from the facts that $v \cap E_1 = v_1$, and $(I - v) \cap E_1$ is a subset of $(I_1 - v_1)$.)

Case 2 $e \in v \cap O_1 \cap I_2$

Since $\gamma_2 \upharpoonright v = (\langle e\rangle^\wedge\gamma) \upharpoonright E_2$, we can write γ_2 in the form $\gamma'_{2,1}{}^\wedge\langle e\rangle^\wedge\gamma'_{2,2}$, where $\gamma'_{2,1} \upharpoonright v_2 = \langle\rangle$, and $\gamma'_{2,2} \upharpoonright v_2 = \gamma \upharpoonright E_2$. Since $\gamma'_{2,1} \upharpoonright (I_2 - v_2) = \langle\rangle$, by definition of *OK_SO_FAR*, and $\gamma'_{2,1} \upharpoonright v_2 = \langle\rangle$, then we have $\gamma'_{2,1} \upharpoonright I_2 = \langle\rangle$. Thus $\gamma'_{2,1}$ consists of outputs and internal events for system ES_2. Since every event common to systems ES_1 and ES_2 must be an input for one system and an output for the other, we can conclude that $\gamma'_{2,1} \upharpoonright E_1$, which is the subsequence of $\gamma'_{2,1}$ containing events common to the two subsystems, must consist of only inputs for ES_1. Since it contains no events in v, it must contain events in $I_1 - v_1$.

Now, we have

$$\begin{aligned}
(\alpha_1 \upharpoonright E_1)^\wedge\gamma_1 &\in T_1 \\
\gamma_1 \upharpoonright (I_1 - v_1) &= \langle\rangle \\
\gamma'_{2,1} \upharpoonright E_1 &\in I_1^* \\
(\gamma'_{2,1} \upharpoonright E_1) \upharpoonright v_1 &= \langle\rangle \upharpoonright v_1 \equiv \langle\rangle
\end{aligned}$$

Therefore, since v_1 is restrictive for system ES_1, there must be a $\gamma'_1 \in E_1^*$ such that

$$\begin{aligned}
(\alpha_1 \upharpoonright E_1)^\wedge(\gamma'_{2,1} \upharpoonright E_1)^\wedge\gamma'_1 &\in T_1 \\
\gamma'_1 \upharpoonright (I_1 - v_1) &= \langle\rangle \\
\gamma'_1 \upharpoonright v_1 &= \gamma_1 \upharpoonright v_1
\end{aligned}$$

Since $\gamma_1 \upharpoonright v_1 = \gamma_1 \upharpoonright v = \langle e\rangle^\wedge(\gamma \upharpoonright E_1)$, it must be that γ'_1 can be written as $\gamma_{1,1}{}^\wedge\langle e\rangle^\wedge\gamma_{1,2}$, where $\gamma_{1,1} \upharpoonright v_1 = \langle\rangle$, and $\gamma_{1,2} \upharpoonright v_1 = \gamma \upharpoonright E_1$. By similar arguments as in the last paragraph, we can conclude that $(\gamma_{1,1}{}^\wedge\langle e\rangle) \upharpoonright E_2$ is an element of I_2^*, and $(\gamma_{1,1}{}^\wedge\langle e\rangle) \upharpoonright v_2 = \langle e\rangle$.

Turning back to system ES_2, we have

$$\begin{aligned}
(\alpha_1 \upharpoonright E_2)^\wedge\gamma'_{2,1}{}^\wedge\langle e\rangle^\wedge\gamma'_{2,2} &\in T_2 \\
\langle e\rangle &\in I_2^* \\
(\gamma_{1,1}{}^\wedge\langle e\rangle) \upharpoonright E_2 &\in I_2^* \\
((\gamma_{1,1}{}^\wedge\langle e\rangle) \upharpoonright E_2) \upharpoonright v_2 &= \langle e\rangle \upharpoonright v_2 = \langle c\rangle \\
\gamma'_{2,2} \upharpoonright (I_2 - v_2) &= \langle\rangle
\end{aligned}$$

Therefore, since v_2 is restrictive for system ES_2, there is a $\gamma_{2,2} \in E_2^*$ such that

$$\begin{aligned}
(\alpha_1 \upharpoonright E_2)^\wedge\gamma'_{2,1}{}^\wedge((\gamma_{1,1}{}^\wedge\langle e\rangle) \upharpoonright E_2)^\wedge\gamma_{2,2} &\in T_2 \\
\gamma_{2,2} \upharpoonright (I_2 - v_2) &= \langle\rangle \\
\gamma_{2,2} \upharpoonright v_2 &= \gamma'_{2,2} \upharpoonright v_2 = \gamma \upharpoonright E_2
\end{aligned}$$

If we let α_2 be $\gamma'_{2,1}{}^\wedge\gamma_{1,1}{}^\wedge\langle e\rangle$, then we can see that $\langle\alpha_1{}^\wedge\alpha_2, \gamma_{1,2}, \gamma_{2,2}, \gamma\rangle$ is an element of OK_SO_FAR, and $\alpha_2 \upharpoonright v = \langle e\rangle$, and $\alpha_2 \upharpoonright (I - v) = \langle\rangle$, as claimed. □

Theorem 3 *Let* $\langle \alpha, \gamma_1, \gamma_2, \gamma \rangle$ *be an element of* OK_SO_FAR. *Then for some* $\alpha' \in E^*$, $\gamma_1' \in E_1^*$, *and* $\gamma_2' \in E_2^*$

$$\begin{aligned} \langle \alpha \hat{} \alpha', \gamma_1', \gamma_2', \langle\rangle \rangle &\in \text{OK\_SO\_FAR} \\ \alpha' \upharpoonright v &= \gamma \\ \alpha' \upharpoonright (I - v) &= \langle\rangle \end{aligned}$$

This theorem says that we can eventually move all the events from the fourth component of an element of *OK_SO_FAR* into the first component.

Proof:

We prove the theorem by induction on the length of γ.

Base Case : $\gamma = \langle\rangle$

In this case we can let $\alpha' = \langle\rangle$, $\gamma_1' = \gamma_1$, and $\gamma_2' = \gamma_2$.

Inductive Hypothesis: For all $\alpha_0 \in E^*$, $\gamma_{1,0} \in E_1^*$, $\gamma_{2,0} \in E_2^*$, and $\gamma_0 \in v^*$, if

- the length of γ_0 is less than or equal to n
- $\langle \alpha_0, \gamma_{1,0}, \gamma_{2,0}, \gamma_0 \rangle$ is an element of *OK_SO_FAR*

then for some $\alpha_1 \in E^*$, $\gamma_{1,1} \in E_1^*$, and $\gamma_{2,1} \in E_2^*$,

- $\langle \alpha_0 \hat{} \alpha_1, \gamma_{1,1}, \gamma_{2,1}, \langle\rangle \rangle$ is an element of *OK_SO_FAR*
- $\alpha_1 \upharpoonright v = \gamma_0$
- $\alpha_1 \upharpoonright (I - v) = \langle\rangle$

Inductive Step: If the length of γ is $n+1$, then we can write γ as $\langle e \rangle ^\wedge \gamma_0$, where the length of γ_0 is n. Then we have that $\langle \alpha, \gamma_1, \gamma_2, \langle e \rangle ^\wedge \gamma_0 \rangle$ is in *OK_SO_FAR*. By Theorem 2, there exist $\alpha'_0 \in E^*$, $\gamma'_{1,0} \in E_1^*$, and $\gamma'_{2,0} \in E_2^*$ such that $\langle \alpha ^\wedge \alpha'_0, \gamma'_{1,0}, \gamma'_{2,0}, \gamma_0 \rangle$ is in *OK_SO_FAR*, $\alpha'_0 \uparrow v = \langle e \rangle$, and $\alpha \uparrow (I - v) = \langle \rangle$. Now, since the length of γ_0 is n, we can apply the inductive hypothesis, which gives us the existence of $\alpha'_1 \in E^*$, $\gamma'_1 \in E_1^*$, and $\gamma'_2 \in E_2^*$ such that

- $\langle \alpha ^\wedge \alpha'_0 {}^\wedge \alpha'_1, \gamma'_1, \gamma'_2, \langle \rangle \rangle$ is in *OK_SO_FAR*
- $\alpha'_1 \uparrow v = \gamma_0$
- $\alpha'_1 \uparrow (I - v) = \langle \rangle$

Thus, letting $\alpha' \equiv \alpha'_0 {}^\wedge \alpha'_1$, we have

- $\langle \alpha ^\wedge \alpha', \gamma'_1, \gamma'_2, \langle \rangle \rangle \in$ *OK_SO_FAR*
- $\alpha' \uparrow v = \langle e \rangle ^\wedge \gamma_0 = \gamma$
- $\alpha' \uparrow (I - v) = \langle \rangle$

This completes the proof. □

Theorem 4 *Let* $\langle \alpha, \gamma_1, \gamma_2, \gamma \rangle$ *be an element of* OK_SO_FAR. *Then for some* $\alpha' \in E^*$,

- $\langle \alpha ^\wedge \alpha', \langle \rangle, \langle \rangle, \langle \rangle \rangle$ *is in* OK_SO_FAR
- $\alpha' \uparrow v = \gamma$
- $\alpha' \uparrow (I - v) = \langle \rangle$

This theorem says that we can eventually eliminate all the events in the last three components of an element of *OK_SO_FAR*. Since these components told us what we still had left to accomplish towards merging traces of the constituent systems into traces of the composite system, this theorem says that we can eventually succeed, and that the first component is the merged composite trace.

Proof:

By Theorem 3, for some $\alpha' \in E^*, \gamma_1' \in E_1^*, \gamma_2' \in E_2^*$, $\langle \alpha^\wedge\alpha', \gamma_1', \gamma_2', \langle\rangle\rangle$ is an element of *OK_SO_FAR*, and $\alpha' \uparrow v = \gamma$, and $\alpha' \uparrow (I - v) = \langle\rangle$. Then it is sufficient to show that $\langle \alpha^\wedge\alpha', \langle\rangle, \langle\rangle, \langle\rangle\rangle$ is also in *OK_SO_FAR*.

- $(\alpha^\wedge\alpha'^\wedge\langle\rangle) \uparrow E_1$ is in T_1, since $(\alpha^\wedge\alpha'^\wedge\gamma_1')$ is, and the set of traces is closed under taking initial segments
- $\langle\rangle \uparrow v = \langle\rangle \uparrow E_1$
- $(\alpha^\wedge\alpha'^\wedge\langle\rangle) \uparrow E_2$ is in T_2
- $\langle\rangle \uparrow v = \langle\rangle \uparrow E_2$
- $\langle\rangle \uparrow (I_1 - v_1) = \langle\rangle$
- $\langle\rangle \uparrow (I_2 - v_2) = \langle\rangle$

We can see that $\langle \alpha^\wedge\alpha', \langle\rangle, \langle\rangle, \langle\rangle\rangle$ is an element of *OK_SO_FAR*. □

Theorem 5 *With* ES$_1$, ES$_2$, ES, v, v_1, *and* v_2 *as above,* v *is a restrictive view of system* ES.

Proof:

We need to show that for all $\alpha \in E_1^*$, $\beta \in I^*$, $\beta' \in I^*$, and $\gamma \in E^*$, if $\beta \upharpoonright v = \beta' \upharpoonright v$, and $\alpha^\wedge\beta^\wedge\gamma$ is in T, and $\gamma \upharpoonright (I - v) = \langle\rangle$, then there exists a $\gamma' \in E^*$ such that

$$\begin{array}{rcl} \gamma' \upharpoonright v & = & \gamma \upharpoonright v \\ \gamma' \upharpoonright (I - v) & = & \langle\rangle \\ \alpha^\wedge\beta'^\wedge\gamma' & \in & T \end{array}$$

Let α, β, β', and γ be sequences meeting the above conditions. Since $\alpha^\wedge\beta^\wedge\gamma$ is in T, then $(\alpha^\wedge\beta^\wedge\gamma) \upharpoonright E_1$ is in T_1, and $(\alpha^\wedge\beta^\wedge\gamma) \upharpoonright E_2$ is in T_2.

Let us define the following sequences:

- $\alpha_1 \equiv \alpha \upharpoonright E_1$
- $\beta_1 \equiv \beta \upharpoonright E_1$
- $\beta_1' \equiv \beta' \upharpoonright E_1$
- $\gamma_1 \equiv \gamma \upharpoonright E_1$
- $\alpha_2 \equiv \alpha \upharpoonright E_2$
- $\beta_2 \equiv \beta \upharpoonright E_2$
- $\beta_2' \equiv \beta' \upharpoonright E_2$
- $\gamma_2 \equiv \gamma \upharpoonright E_2$

Then we have

$$\begin{array}{rcl} \alpha_1{}^\wedge\beta_1{}^\wedge\gamma_1 & \in & T_1 \\ \beta_1 & \in & I_1^* \\ \beta_1' \upharpoonright v_1 & = & \beta_1 \upharpoonright v_1 \end{array}$$

Because v_1 is restrictive, we can use Theorem 1, with the following instantiations:

- $\alpha \mapsto \alpha_1{}^\wedge\beta_1$
- $\gamma_1 \mapsto \gamma_1$

Thus there exists a $\gamma_1' \in E_1^*$ such that

$$\begin{aligned} \alpha_1{}^\wedge\beta_1{}^\wedge\gamma_1' &\in T_1 \\ \gamma_1' \upharpoonright v_1 &= \gamma_1 \upharpoonright v_1 \\ \gamma_1' \upharpoonright (I_1 - v_1) &= \langle\rangle \end{aligned}$$

Now we can use the definition of a restrictive view, definition 12 with the instantiations

- $\alpha \mapsto \alpha_1$
- $\beta_1 \mapsto \beta_1$
- $\beta_2 \mapsto \beta_1'$
- $\gamma_1 \mapsto \gamma_1'$

to get a $\gamma_1'' \in E_1^*$ such that

$$\begin{aligned} \alpha_1{}^\wedge\beta_1'{}^\wedge\gamma_1'' &\in T_1 \\ \gamma_1'' \upharpoonright v_1 &= \gamma_1' \upharpoonright v_1 \equiv \gamma_1 \upharpoonright v_1 \\ \gamma_1'' \upharpoonright (I_1 - v_1) &= \langle\rangle \end{aligned}$$

Likewise, there is a $\gamma_2'' \in E_2^*$ such that

$$\begin{aligned} \alpha_2{}^\wedge\beta_2'{}^\wedge\gamma_2'' &\in T_2 \\ \gamma_2'' \upharpoonright v_2 &= \gamma_2 \upharpoonright v_2 \\ \gamma_2'' \upharpoonright (I_2 - v_2) &= \langle\rangle \end{aligned}$$

Checking the definition of *OK_SO_FAR*, we can see that $\langle \alpha{}^\wedge\beta', \gamma_1'', \gamma_2'', \gamma \upharpoonright v\rangle$ is in *OK_SO_FAR*. By theorem 4, there is a $\gamma' \in E^*$ such that

$$\begin{aligned} \langle \alpha{}^\wedge\beta'{}^\wedge\gamma', \langle\rangle, \langle\rangle, \langle\rangle\rangle &\in \mathit{OK\_SO\_FAR} \\ \gamma' \upharpoonright v &= \gamma \upharpoonright v \\ \gamma' \upharpoonright (I - v) &= \langle\rangle \end{aligned}$$

Since $(\alpha^\wedge\beta'^\wedge\gamma') \upharpoonright E_1$ is an element of T_1, and $(\alpha^\wedge\beta'^\wedge\gamma') \upharpoonright E_2$ is an element of T_2, then by the definition of a simple hook-up, definition 3, $\alpha^\wedge\beta'^\wedge\gamma'$ is an element of T. Thus γ' is the desired sequence needed for our proof. □

9.8 Hook-Up Security for Rated Event Systems

Definition 14 A rated event system $RES = \langle E, I, O, T, L, \leq, lvl \rangle$ will be said to be *hook-up secure* if for all $l \in L$, $vue(l)$ is restrictive.

Definition 15 We will say that *RES* has the *Goguen-Meseguer noninterference property*[1] (or *G-M property* for short) if the set of traces T is deterministic, and for all $l \in L$ and for all $t \in T$, there is a $t' \in T$ such that

- $t' \upharpoonright vue(l) = t \upharpoonright vue(l)$
- $t' \upharpoonright (I - vue(l)) = \langle\rangle$

Theorem 6 *If* RES *is hook-up secure, and its set of traces T is deterministic, then it has the G-M property.*

Proof:

This follows immediately from Corollary 1. □

Theorem 7 *If* RES *is the simple rated hook-up of component systems* RES_1 *and* RES_2, *and the components are hook-up secure, then* RES *is hook-up secure.*

[1] In [Sut 86] is a discussion connecting Goguen-Meseguer with information flow.

Proof:

Let RES be the event system $\langle E, I, O, T, L, \leq, lvl \rangle$, RES_1 the event system $\langle E_1, I_1, O_1, T_1, L, \leq, lvl_1 \rangle$, and RES_2 the system $\langle E_2, I_2, O_2, T_2, L, \leq, lvl_2 \rangle$. Let l be a level in L. In definition 10 $vue(l)$ was defined to be

$$\{e \in E \mid lvl(e) \leq l\}$$

Likewise, $vue_1(l)$ and $vue_2(l)$ are the sets

$$\{e \in E_1 \mid lvl_1(e) \leq l\}$$

and

$$\{e \in E_2 \mid lvl_2(e) \leq l\}$$

respectively.

It is then obvious that for all $l \in L$,

$$vue_1(l) = vue(l) \cap E_1$$

and

$$vue_2(l) = vue(l) \cap E_2$$

Since RES_1 and RES_2 are both hook-up secure, for all $l \in L$, $vue_1(l)$ and $vue_2(l)$ are restrictive for systems RES_1 and RES_2, respectively. Using the results of Theorem 5, for all $l \in L$, $vue(l)$ is restrictive for system RES. Thus RES is hook-up secure. □

Chapter 10

An Example : A Delay Queue

In this chapter we illustrate the concept of hook-up security through a simple example, an unbounded delay queue. Even for this simple example the proof of hook-up security is not trivial. It is hoped, however, that the proof of security of a large system can be divided up via the hook-up theorem into proofs involving components not much more complicated than the delay queue. (The security of the more realistic *bounded* queue is more problematic, because of the possibility of information flows through the error messages indicating that the queue is full.)

10.1 Description of the Delay Queue

A delay queue is simply a pipeline for carrying messages from one point in a system to another. There is no command to release messages; the messages are released as soon as they arrive. Thus the effect is to put a delay between the sending and receiving of a message. In our simple model, we assume that the delay can be absolutely arbitrary, subject to the restriction that messages come off the queue in the same order that they went on.

We formalize the model by introducing the set of messages, the input and output events built from them, and the set of traces describing the possible behaviors of the queue:

$$\begin{array}{lcl} M & = & \text{the set of messages} \\ I \equiv \{\text{ in }\} \times M & = & \text{the input events} \\ O \equiv \{\text{ out }\} \times M & = & \text{the output events} \\ E \equiv I \cup O & = & \text{the set of events} \end{array}$$

A trace of the delay queue is any sequence of events such that each output event follows its corresponding input event, and the outputs come in the same order as the corresponding inputs. This can be formalized by defining the function *mp* which maps a sequence of input events to the corresponding complete sequence of output events. We will actually define *mp* so that it is defined on all sequences of events, whethei they include inputs or outputs or both, except that it will ignore any output in computing the result.

$$\begin{array}{lcl} mp(\langle\rangle) & = & \langle\rangle \\ mp(\alpha^\wedge\langle\langle in, m\rangle\rangle) & = & mp(\alpha)^\wedge\langle\langle out, m\rangle\rangle \\ mp(\alpha^\wedge\langle\langle out, m\rangle\rangle) & = & mp(\alpha) \end{array}$$

The set of traces T is then defined by:

$$T \equiv \{\ \tau \in E^* \mid \forall \tau_1 \sqsubseteq \tau\ [\ \tau_1 \uparrow O \sqsubseteq mp(\tau_1)\ ]\ \}$$

This states that the outputs must always lag behind the inputs, so that the outputs are always an initial segment of the mapped inputs. It is easy to see that this set of traces has the following properties:

1. If $\alpha^\wedge\beta$ is a trace, then so is α.

2. If α is a trace, and x is an input, then $\alpha^\wedge\langle x\rangle$ is a trace.

Thus it meets the requirements that it be input-total and closed under initial segments.

10.2 Useful Facts

We need a few facts before we start proving that the delay queue is hook-up secure.

Claim 1 The function *mp* distributes over concatenation:

$$mp(\alpha{}^\wedge\beta) = mp(\alpha){}^\wedge mp(\beta)$$

Claim 2 Restriction distributes over concatenation:

$$(\alpha{}^\wedge\beta) \upharpoonright E_1 = (\alpha \upharpoonright E_1){}^\wedge(\beta \upharpoonright E_1)$$

Claim 3 If two sequences α_1 and α_2 are both initial segments of a third sequence β, then they are comparable (one is an initial segment of the other):

$$\alpha_1 \sqsubseteq \beta \;\&\; \alpha_2 \sqsubseteq \beta \quad\Rightarrow\quad \alpha_1 \sqsubseteq \alpha_2 \vee \alpha_2 \sqsubseteq \alpha_1$$

Claim 4 It is permissible to "add" and "cancel" equal sequences from both sides of an initial segment relation:

$$\begin{array}{lcl} \beta_1 \sqsubseteq \beta_2 & \Rightarrow & \alpha{}^\wedge\beta_1 \sqsubseteq \alpha{}^\wedge\beta_2 \\ \alpha{}^\wedge\beta_1 \sqsubseteq \alpha{}^\wedge\beta_2 & \Rightarrow & \beta_1 \sqsubseteq \beta_2 \end{array}$$

Claim 5 If $\alpha{}^\wedge\beta_1 \sqsupset \gamma$ and $\alpha \sqsubseteq \gamma$, then there is an "intermediate value" β_2 such that $\beta_2 \sqsubset \beta_1$ and $\alpha{}^\wedge\beta_2 = \gamma$.

Claim 6 $\alpha \sqsupset \beta$ if and only if for some α_1 and e, $\alpha = \alpha_1{}^\wedge\langle e\rangle$ and $\alpha_1 \sqsupseteq \beta$.

Claim 7 The relation $\sqsupseteq$ is reflexive, transitive, and anti-symmetric. This means formally that:

$$\begin{array}{l} \alpha \sqsupseteq \alpha \\ \alpha \sqsupseteq \beta \;\;\&\;\; \beta \sqsupseteq \gamma \;\;\Rightarrow\;\; \alpha \sqsupseteq \gamma \\ \alpha \sqsupseteq \beta \;\;\&\;\; \beta \sqsupseteq \alpha \;\;\Rightarrow\;\; \alpha = \beta \end{array}$$

Claim 8 If τ is a trace in T, and γ is an element of E^*, and for all γ' such that $\langle\rangle \sqsubseteq \gamma' \sqsubseteq \gamma$:

$$mp(\tau^\wedge\gamma') \sqsupseteq (\tau^\wedge\gamma') \uparrow O$$

then $\tau^\wedge\gamma$ is a trace in T.

The above claims are easy to prove by simple algebra and by induction on the length of sequences.

10.3 Security of the Delay Queue

Theorem 8 For any set $M_1 \subseteq M$ of messages, the view $V_1 \equiv \{\text{in}\} \times M_1 \cup \{\text{out}\} \times M_1$ is restrictive. This means that for any $\alpha, \gamma \in E^*$, $\beta, \beta' \in I^*$, if

$$\alpha^\wedge\beta^\wedge\gamma \in T \;\&\; \beta \uparrow V_1 = \beta' \uparrow V_1 \;\&\; \gamma \uparrow (I - V_1) = \langle\rangle$$

then for some $\gamma' \in E^*$

$$\alpha^\wedge\beta'^\wedge\gamma' \in T \;\&\; \gamma' \uparrow V_1 = \gamma \uparrow V_1 \;\&\; \gamma' \uparrow (I - V_1) = \langle\rangle$$

We prove the theorem by means of the following lemmas:

Lemma 1 If β' is formed from β by deleting an input $\langle \text{in}, m\rangle$ where m is not in M_1, then there is a γ' with the desired properties.

Lemma 2 If β' is formed from β by adding an input $\langle \text{in}, m\rangle$ where m is not in M_1, then there is a γ' with the desired properties.

10.3.1 Proof of Lemma 1

Consider β of the form $\beta_1{}^\wedge\langle e\rangle{}^\wedge\beta_2$ and β' of the form $\beta_1{}^\wedge\beta_2$, where $e = \langle \mathrm{in}, m\rangle$. Let τ be the trace $\alpha{}^\wedge\beta_1{}^\wedge\langle e\rangle{}^\wedge\beta_2{}^\wedge\gamma$, and let τ' be formed from τ by deleting the event e; $\tau' = \alpha{}^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma$, which is equal to $\alpha{}^\wedge\beta'{}^\wedge\gamma$.

We then have two cases to consider:

1. τ' is a trace
2. τ' is not a trace

In the first case, we are done; we can let γ' be equal to γ. We then have:

- $\tau' = \alpha{}^\wedge\beta'{}^\wedge\gamma'$
- τ' is a trace
- $\gamma' \uparrow V_1 = \gamma \uparrow V_1$
- $\gamma' \uparrow (I - V_1) = \langle\rangle$ (since by assumption $\gamma \uparrow (I - V_1) = \langle\rangle$)

which is what we needed to prove.

In the second case, we have that τ' is not a trace, but the initial segment $\alpha{}^\wedge\beta_1$ is a trace (since it is an initial segment of the trace τ, as well, and traces are closed under initial segments). Thus we have the following facts:

- $mp(\alpha{}^\wedge\beta_1{}^\wedge\langle e\rangle{}^\wedge\beta_2{}^\wedge\gamma) \sqsupseteq (\alpha{}^\wedge\beta_1{}^\wedge\langle e\rangle{}^\wedge\beta_2{}^\wedge\gamma) \uparrow O$ (this is implied by the definition of the set of traces T.)
- $mp(\alpha{}^\wedge\beta_1{}^\wedge\langle e\rangle{}^\wedge\beta_2{}^\wedge\gamma) \sqsupseteq mp(\alpha{}^\wedge\beta_1)$ (because mp is distributive.)

By claim 3, two sequences which are both initial segments of a third sequence must be comparable. Thus we have two subcases:

1. $mp(\alpha^\wedge\beta_1) \sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma) \uparrow O$
2. $mp(\alpha^\wedge\beta_1) \sqsubset (\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma) \uparrow O$

In the first subcase, we have that

- $\alpha^\wedge\beta_1$ is a trace.
- $\alpha^\wedge\beta_1{}^\wedge\beta_2$ is a trace, since β_2 is a sequence of inputs and traces can always be extended by inputs.
- $mp(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \sqsupseteq mp(\alpha^\wedge\beta_1)$ for any δ, since mp is distributive.
- $mp(\alpha^\wedge\beta_1) \sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma) \uparrow O$ by assumption.
- $(\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma) \uparrow O \sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\delta) \uparrow O$ for any $\delta \sqsubseteq \gamma$, since the operation $\uparrow O$ is distributive.
- $mp(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\delta) \uparrow O$ for $\delta \sqsubseteq \gamma$, since $\sqsupseteq$ is transitive.
- $(\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\delta) \uparrow O = (\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \uparrow O$ since e is not an output, and so $\langle e\rangle \uparrow O = \langle\rangle$.
- $mp(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \uparrow O$ for $\delta \sqsubseteq \gamma$ by substitution.

So we have that

$$\alpha^\wedge\beta_1{}^\wedge\beta_2$$

is a trace, and for all $\delta \sqsubseteq \gamma$,

$$mp(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \uparrow O$$

Therefore,

$$\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma$$

is a trace, by claim 8. But this contradicts the assumption that τ' is not a trace, so the first subcase is vacuous.

Therefore, we are left with the subcase that

$$mp(\alpha^\wedge\beta_1) \sqsubset (\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma) \uparrow O$$

Since $\alpha^\wedge\beta_1$ is a trace (it is an initial segment of a trace), we have by the definition of the set of traces, $mp(\alpha^\wedge\beta_1) \sqsupseteq (\alpha^\wedge\beta_1) \uparrow O$. Thus $mp(\alpha^\wedge\beta_1)$ is bounded above by $(\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma) \uparrow O$, and below by $(\alpha^\wedge\beta_1) \uparrow O$, which is equal to $(\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2) \uparrow O$ (because e is an input, and β_2 is an input sequence, they disappear when the operation $\uparrow O$ is performed.) By claim 5 there is a $\gamma_1 \sqsubset \gamma$ such that

$$mp(\alpha^\wedge\beta_1) = (\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma_1) \uparrow O$$

which is equivalent to

$$mp(\alpha^\wedge\beta_1) = (\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1) \uparrow O \tag{10.1}$$

because e is an input.

We will choose γ_1 to be the largest initial segment of γ obeying equation 10.1. Since $\gamma_1 \sqsubset \gamma$, we can write γ as $\gamma_1{}^\wedge\langle e'\rangle^\wedge\gamma_2$ for some e' and γ_2. It must be the case that e' is an output; otherwise, $(\gamma_1 \uparrow e') \uparrow O$ would be equal to $\gamma_1 \uparrow O$, contradicting the assumption that γ_1 is the *largest* initial segment of γ obeying equation 10.1.

Since $\alpha^\wedge\beta_1$ is a trace, then so is $\alpha^\wedge\beta_1{}^\wedge\beta_2$, because β_2 is an input sequence and traces can always be extended by inputs. Furthermore, for any $\delta \sqsubseteq \gamma_1$,

$$mp(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \sqsupseteq mp(\alpha^\wedge\beta_1)$$

and also we know that

$$(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1) \uparrow O \;\sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\delta) \uparrow O$$

Therefore, by claim 8 and by equation 10.1, $\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1$ is a trace.

Now, since $\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma_1{}^\wedge\langle e'\rangle^\wedge\gamma_2$ is a trace, all initial segments are also, so therefore, for any $\delta \sqsubseteq \gamma_2$, we have

$$mp(\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma_1{}^\wedge\langle e'\rangle^\wedge\delta) \sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma_1{}^\wedge\langle e'\rangle^\wedge\delta) \uparrow O$$

Using the fact that mp and $\uparrow O$ are distributive, we can write this as:

$$mp(\alpha^\wedge\beta_1)^\wedge mp(\langle e\rangle)^\wedge mp(\beta_2{}^\wedge\gamma_1{}^\wedge\langle e'\rangle^\wedge\delta)$$

$$\sqsupseteq$$

$$[(\alpha^\wedge\beta_1{}^\wedge\langle e\rangle^\wedge\beta_2{}^\wedge\gamma_1) \uparrow O]^\wedge[\langle e'\rangle \uparrow O]^\wedge[\delta \uparrow O]$$

Because e is not an output, it is removed by the operation $\uparrow C$. We can then rewrite the right side of the inequality as

$$[(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1) \uparrow O]^\wedge[\langle e'\rangle \uparrow O]^\wedge[\delta \uparrow O]$$

By equation 10.1we know that $(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1) \uparrow O$ is equal to $mp(\alpha^\wedge\beta_1)$. Therefore, we can rewritc the right side once again as

$$[mp(\alpha^\wedge\beta_1) \uparrow O]^\wedge[\langle e'\rangle \uparrow O]^\wedge[\delta \uparrow O]$$

Thus we have:

$$mp(\alpha^\wedge\beta_1)^\wedge mp(\langle e\rangle)^\wedge mp(\beta_2{}^\wedge\gamma_1{}^\wedge\langle e'\rangle^\wedge\delta) \sqsupseteq [mp(\alpha^\wedge\beta_1) \uparrow O]^\wedge[\langle e'\rangle \uparrow O]^\wedge[\delta \uparrow O]$$

so by equation 10.1, and claim 4

$$mp(\langle e\rangle)^\wedge mp(\beta_2{}^\wedge\gamma_1{}^\wedge\langle e'\rangle^\wedge\delta) \sqsupseteq [\langle e'\rangle \uparrow O]^\wedge[\delta \uparrow O]$$

Now since e is an input, $mp(\langle e\rangle)$ is a singleton sequence. Also, since e' is an output, $\langle e'\rangle \uparrow O$ is equal to $\langle e'\rangle$. Thus the only way that the above inequality can hold is if the following are true:

- $mp(\langle e \rangle) = \langle e' \rangle$
- $mp(\beta_2{}^\wedge\gamma_1{}^\wedge\langle e' \rangle^\wedge\delta) \sqsupseteq \delta \uparrow O$

Since e' is an output, it has no effect on the result of operating with mp. Therefore:

$$mp(\beta_2{}^\wedge\gamma_1{}^\wedge\delta) \sqsupseteq \delta \uparrow O$$

Concatenating $mp(\alpha^\wedge\beta_1) = (\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1) \uparrow O$ to both sides of the inequality, as equation 10.1 permits us to do, yields:

$$mp(\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1{}^\wedge\delta) \sqsupseteq (\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1{}^\wedge\delta) \uparrow O$$

Since this is true for any $\delta \sqsubseteq \gamma_2$, we conclude from claim 8 that

$$\alpha^\wedge\beta_1{}^\wedge\beta_2{}^\wedge\gamma_1{}^\wedge\gamma_2$$

is a trace.

Thus we can let γ' be $\gamma_1{}^\wedge\gamma_2$. This only differs from γ by the absence of the event e'. However, since we have established that $\langle e' \rangle = mp(\langle e \rangle)$, and since e was by assumption in $I - V_1$, then e must be of the form $\langle in, m \rangle$ for some $m \notin M_1$, and e' must be of the form $\langle out, m \rangle$, and so e' is not in V_1, either. Therefore, $\gamma' \uparrow V_1 = \gamma \uparrow V_1$. This completes the proof of the first lemma.

10.3.2 Proof of Lemma 2

The proof of this lemma is essentially the same as the proof of the last lemma, except that in this case it is necessary to show that inputs can be added to a trace, rather than deleted.

10.3.3 Proof of Theorem 8

Using the special cases in which the new input sequence β' differs from β by a single input (either deleted or added), we can prove the case for β and β' differing in arbitrary ways by induction. It is obvious that any change between two traces can be broken down into a sequence of changes that only involve one event.

Chapter 11

A State Machine Formulation of Restrictiveness

As we saw in the last chapter, proving security directly on the set of traces associated with a system can be tedious, even for simple systems. If the traces are produced by the actions of a machine, it is often easier to prove properties about the state transitions of the machine than it is to prove properties about the traces of the machine. In this chapter, we introduce a property of state machines, which we call *state machine restrictiveness*, such that if a state machine is restrictive with respect to a particular view, then the event system associated with its set of traces is restrictive with respect to that view. This gives rise to a definition of hook-up security for state machines that does not rely on first computing the set of traces.

A *state machine* is an abstract model of an information processing system. In this model, the information possessed by the system is recorded in the *state* of the system. Information is processed through a sequence of *state transitions* made by the system as it changes state to reflect progress in the task being performed or to reflect new information or requests fed into the system from outside. The system may also change state as a result of sending out replies, status information, or requests of its own; in which case the state records the system's place in the communication protocol. We can thus distinguish three kinds of state transitions:

- The machine receives new information from the outside in the form of an *input* event, and changes state to record this new information.
- The machine sends out information in the form of an *output event*, and changes state to reflect the place in the protocol.
- The machine makes an *internal transition*; it changes state to reflect internal progress in the information processing.

11.1 Attributes of a State Machine

To describe a state machine, it is necessary to give the following components:

1. The set S of possible *states*. The state must record all information the system needs to know how to respond in the future, which includes the data being processed as well as the place in the communication protocol.
2. The *initial state* σ_0.
3. The set E of possible *events*. The events are signals associated with state transitions; either inputs from outside, outputs sent to outside, or internal transitions.
4. The set I of *inputs*.
5. The set O of *outputs*.
6. The set TR of possible *transitions*. A transition is a state change with the associated signal. Thus a transition has three components: the state before the transition, the state after the transition, and the event (input, output, or internal event) accompanying the transition.

We will use the following notation for transitions: If from state σ_1, the system may make a transition to σ_2 with accompanying signal e, then we will indicate this by $\sigma_1 \xrightarrow{e} \sigma_2$.

This can be read as "σ_1 goes to σ_2 on e". We will also have occasion to talk about *extended transitions*: If for some sequence of events $e_1, e_2, ..., e_n$ and some sequence of states $\sigma_1, \sigma_2, ..., \sigma_{n+1}$, we have that $\sigma_1 \xrightarrow{e_1} \sigma_2$ and that $\sigma_j \xrightarrow{e_j} \sigma_{j+1}$ for all $j \leq n$, then we will say that "σ_1 goes to σ_{n+1} on the sequence $\langle e_1, ..., e_n \rangle$", indicated by $\sigma_1 \xrightarrow{\langle e_1, ..., e_n \rangle} \sigma_{n+1}$. By convention, every state goes to itself on the empty sequence: $\sigma_1 \xrightarrow{\langle\rangle} \sigma_1$.

The set of traces of a state machine is the set of event sequences produced by the extended transitions starting from the initial state σ_0.

11.2 Security of State Machines

A user's *projection* on a state machine is what determines how the state machine appears to the user. A projection can be specified by giving two parameters: a set v of events which are visible to the user, and an equivalence relation $\approx$ on states, which tells which states are to be considered identical from the point of view of the user. In a secure system, it is necessary to decide for each user which events that user will be allowed to see and what state information he will be allowed to infer. These considerations give rise to a projection for each user. A user's projection is said to be *restrictive* if it is impossible for the user, through observations visible to the projection, to distinguish states which are considered equivalent for that projection, or to learn anything about the sequence of inputs which are not visible to the projection. For a projection to be restrictive for a system, it is necessary that the transition relation for the system respect the equivalence relation for that projection: the transitions made by the system cannot reveal any state or input information that is forbidden to that projection. The equivalence class of a state under the equivalence relation $\approx$ characterizes the portion of the state that is knowable for the projection.

Let $\sigma_1 \xrightarrow{x} \sigma_1'$ be an arbitrary transition of $\mathcal{S}$. Then for the projection characterized by $\langle v, \approx \rangle$ to be restrictive for state machine $\mathcal{S}$, it must be that:

- If x is an input which is not in view v, then $\sigma_1 \approx \sigma_1'$.

  This rule says that inputs invisible to the projection can have no effect on the state's equivalence class for the projection. If we consider events visible to the projection to be "low-level" and events not visible to the projection to be "high-level", then this is a kind of "no write down" policy—it is not permissible for inputs invisible to a projection to "write down" and affect the information in the state visible to that projection.

Let σ_2 be a state such that $\sigma_1 \approx \sigma_2$. Then there must be a "similar" transition possible from state σ_2. That is, there must be a state σ_2' and a sequence of events γ such that:

- $\sigma_2 \xrightarrow{\gamma} \sigma_2'$
- $\sigma_2' \approx \sigma_1'$
- γ is sufficiently similar to the sequence $\langle x \rangle$.

  For sequence γ to be sufficiently similar to $\langle x \rangle$, it must be that

  - If x is an input, then $\gamma = \langle x \rangle$.
  - If x is a non-input (an output or an internal event) which is not in v, then $\gamma \uparrow (I \cup v) = \langle \rangle$.
  - If x is a non-input and is in v, then γ must be of the form $\gamma_1{}^\wedge\langle x \rangle^\wedge\gamma_2$ where $\gamma_1 \uparrow (I \cup v) = \gamma_2 \uparrow (I \cup v) = \langle \rangle$.

These rules say that the set of ways in which an input may affect a state is equivalent to the set of ways in which that input may affect any equivalent state; also that if an output or internal event is possible from one state, then a similar sequence is possible from any equivalent state, and the sets of possible resulting states are equivalent. Two non-input sequences are similar for the projection if they contain the same visible events occurring in the same order.

The two rules about non-inputs correspond to a "no read up" policy: If two states are equivalent for a projection, then they only differ in "high-level" information (that is, information that is not supposed to be visible to the projection.) By observing outputs, a user can read portions of the state of the system. To prevent him from reading "high-level" information, the visible events in the output sequence can only depend on "low-level" information: the equivalence class of the state for that projection. It is okay to allow any number of invisible outputs and internal events to be interspersed among the visible events, since by assumption the user only sees the visible portion of event sequences.

The rules above imply the trace definition of restrictiveness: If a projection characterized by events v is restrictive for the state machine $\mathcal{S}$, then v will be restrictive for the set of traces produced by $\mathcal{S}$. (There are, however, some state machines which are definitely *not* restrictive, which nevertheless give rise to a restrictive set of traces.) The proof of this fact involves a straightforward induction on the lengths of traces. Rather than proving this fact, and appealing to the hook-up theorem, we will instead prove directly that state machine restrictiveness is a composable property.

11.3 The Composability of Restrictiveness

To see that restrictiveness of state machines is composable, one needs to prove that if a machine $\mathcal{A}$ and machine $\mathcal{B}$ both obey state machine restrictiveness for projections given by $\langle v_A, \approx_A \rangle$ and $\langle v_B, \approx_B \rangle$, respectively, then the composite machine formed by connecting them with communication links obeys state machine restrictiveness for some projection given by $\langle (v_A \cup v_B), \approx \rangle$ for some equivalence relation $\approx$. For the connection to make sense, the two machines must be compatible, in the sense that every shared event must be an output for one machine and input for the other, and the two machines must agree on whether the event is in the composite projection. This last requirement in the case of multi-level security corresponds to the requirement that the two systems agree on the levels of shared events. It is formalized by saying that $v_A \cap E_B = v_B \cap E_A$. We will use v to denote $v_A \cup v_B$.

11.3.1 The Composite Machine

The states of the composite machine are pairs of states of $\mathcal{A}$ with states of $\mathcal{B}$.

The transitions of the composite machine are given by: $\langle A_1, B_1 \rangle \xrightarrow{e} \langle A_2, B_2 \rangle$ only if one of the following cases holds:

1. $A_1 \xrightarrow{e} A_2$ is a legal transition for machine $\mathcal{A}$, and e is not an event of $\mathcal{B}$, and $B_1 = B_2$
2. $B_1 \xrightarrow{e} B_2$ is a legal transition for machine $\mathcal{B}$, and e is not an event of $\mathcal{A}$, and $A_1 = A_2$
3. $A_1 \xrightarrow{e} A_2$ and $B_1 \xrightarrow{e} B_2$ are legal transitions for $\mathcal{A}$ and $\mathcal{B}$, respectively, and e is an input for one machine and an output for the other.

Each transition is either an example of an independent transition of one of the component machines, or is an example of *communication* in which one machine makes an output which is an input to the other machine.

The equivalence relation on the states of the composite machine is obtained from the equivalence relation for the components as follows: $\langle A_1, B_1 \rangle$ is equivalent to $\langle A_2, B_2 \rangle$ if and only if A_1 is equivalent to A_2 and B_1 is equivalent to B_2. Thus $\approx = \approx_A \times \approx_B$.

The inputs of the composite machine are those inputs for either component which are not provided by the other component. Likewise, the outputs of the composite machine are the outputs of either machine which are not inputs to the other.

11.3.2 Demonstrating Restrictiveness for the Composite Machine

We need to show that

1. The composite machine is input-total.

   Each input for the composite machine is an input for one component and is not visible to the other component. Thus according to the transition relation for the composite machine, a legal transition is for the state of the machine that receives the input to change as it would independently, and for the state of the other machine to remain unchanged. Thus for every input there is a corresponding transition, and the composite machine is input-total.

2. If the composite machine receives an input which is not in the projection, then the state remains in the same equivalence class.

   Since an input for the composite machine only affects one of the components, the component *not* affected will trivially remain in the same equivalence class. The component that is affected by the input will remain in the same equivalence class since it makes the same transition it would make in isolation, and by assumption each component is restrictive for the projection.

3. If the composite machine in one state receives an input which is in the projection and makes a transition to a new state, then from a state equivalent to the first state it can make a transition with the same input to a state equivalent to the second state.

   Once again, since only one component is affected, this follows trivially from the fact that each component independently has the property.

4. If a non-input event which is not in the projection can occur in one state of the composite machine resulting in a new state, then from any state equivalent to the first state a sequence of non-inputs invisible to the projection can occur, with the machine winding up in a state equivalent to the second state.

The proof of this is more complicated. Let $\langle A_1, B_1\rangle$ and $\langle A_1, B_1\rangle$ be two equivalent states for the composite machine, so $A_1 \approx_A A_2$ and $B_1 \approx_B B_2$. Let a possible transition be $\langle A_1, B_1\rangle \xrightarrow{e} \langle A_1', B_1'\rangle$.

If e is a non-input for the composite machine then it is a non-input for at least one of the two components. Let us assume that e is a non-input for system $\mathcal{A}$; the other case is proved similarly. Then by definition of the transitions for the composite machine, it must be that $A_1 \xrightarrow{e} A_1'$.

If e is shared by $\mathcal{B}$, then it must be an input by the compatibility conditions. Since it is not in view v, it must be that $B_1' \approx_B B_1$, since by assumption $\mathcal{B}$ is restrictive for view v. If e is not shared by $\mathcal{B}$, then $B_1' = B_1$. Since $\mathcal{A}$ is also restrictive for view v, there must be a state A_2' and a sequence γ of non-inputs with no events in v such that:

$$A_2' \approx_A A_1' \ \& \ A_2 \xrightarrow{\gamma} A_2'$$

Once again by the compatibility conditions, any events in γ which are shared by $\mathcal{B}$ must be inputs. Since they are all events not in view v, then there must be a state B_2' such that:

$$B_2' \approx_B B_2 \ \& \ B_2 \xrightarrow{\gamma\uparrow E_B} B_2'$$

(In fact, if B_2' is *any* state such that $B_2 \xrightarrow{\gamma\uparrow E_B} B_2'$, then $B_2' \approx_B B_2$.)

Thus $\langle A_2, B_2\rangle \xrightarrow{\gamma} \langle A_2', B_2'\rangle$ and $A_2' \approx_A A_1'$ and $B_2' \approx_B B_1'$ (this last equivalence is from transitivity; we established that $B_1 \approx_B B_1'$ and $B_1 \approx_B B_2$ and $B_2 \approx_B B_2'$). By the definition of the equivalence relation on the composite machine, $\langle A_1', B_1'\rangle \approx \langle A_2', B_2'\rangle$.

5. If in one state of the composite machine a non-input occurs which is in view v and the machine makes a transition to a new state, then the composite machine in any state equivalent to the first state can allow a sequence of non-inputs to occur which contains the given event but no other events in the view, and make a transition to a state equivalent to the second state.

This proof is similar to the one above. Once again, let the transition be $\langle A_1, B_1\rangle \xrightarrow{e} \langle A_1', B_1'\rangle$, and let $\langle A_2, B_2\rangle$ be a state equivalent to $\langle A_1, B_1\rangle$. Thus $A_1 \approx_A A_2$ and $B_1 \approx_B B_2$. Also, once again assume that e is a non-input of system $\mathcal{A}$.

By definition of the transitions for the composite machine, it must be that $A_1 \xrightarrow{e} A_1'$. If e is shared by $\mathcal{B}$, then it must be an input by the compatibility conditions, and it must be that $B_1 \xrightarrow{e} B_1'$. Since $\mathcal{A}$ is restrictive, and $A_1 \approx_A A_2$, then there must be a sequence of non-inputs γ and a state A_2' such that

$$A_2' \approx_A A_1' \ \&\ A_2 \xrightarrow{\gamma} A_2'$$

where γ is of the form $\gamma_1{}^\wedge\langle e\rangle^\wedge\gamma_2$ and γ_1 and γ_2 contain only non-inputs not in view v.

By the compatibility conditions, $\gamma_1 \upharpoonright E_B$ contains only inputs for system $\mathcal{B}$ which are not in view v, and similarly for γ_2. Therefore, since $\mathcal{B}$ is restrictive, there is a B_3 such that

$$B_3 \approx_B B_2 \ \&\ B_2 \xrightarrow{\gamma_1 \upharpoonright E_B} B_3$$

Now, by transitivity, $B_3 \approx_B B_1$, so since e is an input to $\mathcal{B}$ and $B_1 \xrightarrow{e} B_1'$, there must be a B_4 such that

$$B_4 \approx_B B_1' \ \&\ B_3 \xrightarrow{e} B_4$$

Since $\gamma_2 \upharpoonright E_B$ contains only inputs for system $\mathcal{B}$ which are not in view v, it must be that for some B_2'

$$B_2' \approx_B B_4 \ \&\ B_4 \xrightarrow{\gamma_2 \upharpoonright E_B} B_2'$$

Thus $\langle A_2, B_2\rangle \xrightarrow{\gamma} \langle A_2', B_2'\rangle$ and $\langle A_2', B_2'\rangle \approx \langle A_1', B_1'\rangle$.

In the case that e is not a shared event, and letting γ be as in the first case, all the events in γ are not in view v_B and so $B_2 \xrightarrow{\gamma \upharpoonright E_B} B_2'$ and $B_2' \approx_B B_2$. In this case, $B_1' = B_1$ and so $B_2' \approx_B B_1'$ also.

11.4 Hook-Up Security of State Machines

In analogy with event systems, we define a *rated state machine* to be a structure $\langle \mathcal{S}, L, \leq, \mathit{lvl} \rangle$, where $\mathcal{S}$ is a state machine, L is a set of security levels, $\leq$ is a partial ordering on L, and *lvl* is a function assigning a security level to each visible (input or output) event of $\mathcal{S}$.

A rated state machine $\langle \mathcal{S}, L, \leq, \mathit{lvl} \rangle$ is defined to be a *hook-up secure* state machine if for each level l there is an equivalence relation $\approx_l$ such that the projection defined by the pair $\langle \mathit{vue}(l), \approx_l \rangle$ is restrictive in state machine $\mathcal{S}$. (The function $\mathit{vue}(l)$ was defined in chapter 9. It is the set of events of level less than or equal to l.)

Chapter 12

The Delay Queue As a State Machine

In this chapter, we redo the proof of the security of the delay queue using the state machine formulation of restrictiveness. The simplicity of this proof illustrates that the state machine formulation is much easier to work with than the trace formulation.

12.1 The Events

As in chapter 10, we will let the inputs and outputs of the delay queue be defined in terms of a set of messages M.

- $I = \text{in} \times M$
- $O = \text{out} \times M$
- $E = I \cup O$

12.2 The States

A state of the delay queue is given by a sequence of messages: the messages that have been received by the queue but have not yet been delivered. The convention we will use is as follows: $\langle m_1, m_2, ..., m_n\langle$ is the state in which the order of messages to be delivered are m_1 then m_2, etc.

12.3 The Transition Relation

There are two kinds of transitions for the delay queue:

1. The queue may receive a message. In this case the state message sequence grows by one message. This corresponds to the transition

   $$\gamma \xrightarrow{\langle \text{in}, m\rangle} \gamma^\wedge\langle m\rangle$$

   The last message to arrive is the last message to be delivered, as messages are delivered starting with the leftmost message.

2. The queue may deliver a message. In this case the state message sequence shrinks by one message. This corresponds to the transition

   $$\langle m\rangle^\wedge\gamma \xrightarrow{\langle \text{out}, m\rangle} \gamma$$

It will be useful for the proof of restrictiveness of the delay queue to define the output function *OF* which takes a sequence of messages and returns the corresponding sequence of outputs. This is defined as follows:

- $OF(\langle\rangle) = \langle\rangle$
- $OF(\sigma^\wedge\langle m\rangle) = OF(\sigma)^\wedge\langle\langle \text{out}, m\rangle\rangle$

The significance of the function OF is this : Starting in a state γ, the queue can make a sequence of outputs (with no inputs occurring) resulting in a state γ' if and only if for some sequence of messages γ'' and some sequence of outputs τ

$$\gamma = \gamma''^\wedge\gamma'$$

and

$$OF(\gamma'') = \tau$$

The function OF distributes over concatenation:

$$OF(\gamma_1{}^\wedge\gamma_2) = OF(\gamma_1)^\wedge OF(\gamma_2)$$

These results are easily proved by induction on the lengths of message sequences.

12.4 The Views and Equivalence Relations

As in chapter 10, we define a view V_1 in terms of a subset M_1 of messages: $V_1 = \text{in} \times M_1 \cup \text{out} \times M_1$. The corresponding equivalence relation $\approx_1$ is defined by

$$\gamma_1 \approx_1 \gamma_2 \quad \leftrightarrow \quad \gamma_1 \uparrow M_1 = \gamma_2 \uparrow M_1$$

With this choice of the views, if $\gamma \uparrow M_1 = \langle\rangle$, then $OF(\gamma) \uparrow V_1 = \langle\rangle$.

12.5 Proving Restrictiveness

To show that the delay queue meets the definition of restrictiveness for state machines, we need to show that:

1. If $m \not\in M_1$, and

$$\gamma \xrightarrow{\langle \mathrm{in}, m \rangle} \gamma'$$

then

$$\gamma' \approx_1 \gamma$$

That is, inputs outside of the view do not change the state's equivalence class.

For any γ, γ' and m as in the hypothesis, we can conclude by the rules for the transition relation that $\gamma' = \gamma^\wedge\langle m \rangle$. Then, by the definition of restriction of a sequence to a set, if m is not in M_1, then $(\gamma^\wedge\langle m \rangle) \upharpoonright M_1 = \gamma \upharpoonright M_1$, and so $\gamma' \approx_1 \gamma$.

2. If $m \in M_1$, and $\gamma_1 \approx_1 \gamma_2$, and

$$\gamma_1 \xrightarrow{\langle \mathrm{in}, m \rangle} \gamma_1'$$

then for some γ_2',

- $\gamma_2 \xrightarrow{\langle \mathrm{in}, m \rangle} \gamma_2'$
- $\gamma_2' \approx_1 \gamma_1'$

For γ_1, γ_1', γ_2, and m as in the hypothesis, we can conclude that γ_1' is of the form $\gamma_1{}^\wedge\langle m \rangle$. Therefore we can let γ_2' be $\gamma_2{}^\wedge\langle m \rangle$. It is then clear that

$$\gamma_2 \xrightarrow{\langle \mathrm{in}, m \rangle} \gamma_2'$$

Since for any sequences σ_1 and σ_2, and any set S,

$$(\sigma_1{}^\wedge\sigma_2) \upharpoonright S = (\sigma_1 \upharpoonright S)^\wedge(\sigma_2 \upharpoonright S)$$

then

$$\gamma_2' \upharpoonright M_1 = (\gamma_2 \upharpoonright M_1)^\wedge(\langle m \rangle \upharpoonright M_1)$$

By assumption, $\gamma_2 \approx_1 \gamma_1$, so by the definition of $\approx_1$, we have

$$\gamma_2' \upharpoonright M_1 = (\gamma_1 \upharpoonright M_1)^\wedge(\langle m \rangle \upharpoonright M_1)$$

The righthand side is equal to $(\gamma_1{}^\wedge\langle m \rangle) \upharpoonright M_1$ or $\gamma_1' \upharpoonright M_1$. Thus $\gamma_2' \approx_1 \gamma_1'$.

3. If $m \not\in M_1$, and $\gamma_1 \approx_1 \gamma_2$, and $\gamma_1 \xrightarrow{\langle \mathrm{out}, m \rangle} \gamma_1'$, then for some γ_2', and some sequence of output events τ

   - $\gamma_2 \xrightarrow{\tau} \gamma_2'$
   - $\tau \upharpoonleft (I \cup V_1) = \langle\rangle$
   - $\gamma_2' \approx_1 \gamma_1'$

   This is the requirement for transitions involving outputs not in view V_1.

   By the transition relation, if $\langle \mathrm{out}, m \rangle$ is an output from γ_1 and the final state is γ_1', then it must be that $\gamma_1 = \langle m \rangle {}^\wedge \gamma_1'$. In this case, if m is not in M_1, then $\gamma_1' \upharpoonleft M_1 = \gamma \upharpoonleft M_1$. By the assumption that $\gamma_1 \approx_1 \gamma_2$, it follows that $\gamma_1' \upharpoonleft M_1 = \gamma_2 \upharpoonleft M_1$, and so $\gamma_1' \approx_1 \gamma_2$. Therefore we can let $\gamma_2' = \gamma_2$ and let $\tau = \langle\rangle$.

4. If $m \in M_1$, and $\gamma_1 \approx_1 \gamma_2$, and $\gamma_1 \xrightarrow{\langle \mathrm{out}, m \rangle} \gamma_1'$, then for some γ_2', and some sequences of output events τ_1 and τ_2

   - $\gamma_2 \xrightarrow{\tau_1 {}^\wedge \langle \langle \mathrm{out}, m \rangle \rangle {}^\wedge \tau_2} \gamma_2'$
   - $\tau_1 \upharpoonleft (I \cup V_1) = \langle\rangle$
   - $\tau_2 \upharpoonleft (I \cup V_1) = \langle\rangle$
   - $\gamma_2' \approx_1 \gamma_1'$

   Once again, it must be that $\gamma_1 = \langle m \rangle {}^\wedge \gamma_1'$. Since $\gamma_1 \approx_1 \gamma_2$, by the definition of $\approx_1$ it must be that $\gamma_2 \upharpoonleft M_1 = \gamma_1 \upharpoonleft M_1$. The righthand side is just $(\langle m \rangle \upharpoonleft M_1) {}^\wedge (\gamma_1' \upharpoonleft M_1)$, which is $\langle m \rangle {}^\wedge (\gamma_1' \upharpoonleft M_1)$, since $m \in M_1$. Therefore, γ_2 must be of the form $\gamma_2'' {}^\wedge \langle m \rangle {}^\wedge \gamma_2'$, where $(\gamma_2' \upharpoonleft M_1) = (\gamma_1' \upharpoonleft M_1)$ and $(\gamma_2'' \upharpoonleft M_1) = \langle\rangle$.

   Therefore, we have that

   $$\gamma_2' \approx_1 \gamma_1'$$

   By the result in section 12.3,

   $$\gamma_2 \xrightarrow{OF(\gamma_2'' {}^\wedge \langle m \rangle)} \gamma_2'$$

Now, $OF(\gamma_2''{}^\wedge\langle m\rangle) = OF(\gamma_2'')^\wedge OF(\langle m\rangle)$ since OF distributes over concatenation. By the definition of OF, $OF(\langle m\rangle) = \langle\langle \text{out}, m\rangle\rangle$. Therefore, we have

$$\gamma_2 \xrightarrow{OF(\gamma_2'')^\wedge\langle\langle \text{out},m\rangle\rangle} \gamma_2'$$

Also by the result in section 12.4, and from the fact that $\gamma_2''\upharpoonright M_1 = \langle\rangle$ and the fact that OF only produces sequences of outputs,

$$OF(\gamma_2'')\upharpoonright(I \cup V_1) = \langle\rangle$$

Thus we can let τ_1 be $OF(\gamma_2'')$ and τ_2 be $\langle\rangle$, and the desired properties hold.

Bibliography

[Ars 85] Arsenault, A. et. al. *Trusted Network Interpretation.* (DOD NCSC-TG-005, National Computer Security Center).

[BLP 76] Bell, D.E. and LaPadula, L.J. *Secure Computer System: Unified Exposition and Multics Interpretation.* (Technical Report no. ESD-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford MA, 1976).

[Dij 76] Dijkstra, E.W. *A Discipline of Programming.* Prentice–Hall, 1976.

[Flo 67] Floyd, R.W. "Assigning Meanings to Programs." *(Proceedings of the Symposium on Applied Mathematics (Mathematical Aspects of Computer Science)),* American Mathematical Society, New York, 1967.

[Gog 84] Goguen, J.A. and Meseguer, J. "Unwinding and Inference Control." *(Proceedings of the 1984 Symposium on Security and Privacy),* IEEE, 1984.

[Goo 86] Good, D.I. et. al. *Report on Gypsy 2.05.* Computational Logic, Austin, TX, 1986.

[Gri 81] Gries, D. *The Science of Programming.* Springer–Verlag, 1981.

[Hoa 69] Hoare, C.A.R. "An Axiomatic Basis of Computer Programming." *Communications of the ACM,* 12:576–583, 1969.

[Hoa 85] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice–Hall, London, 1985.

[Jac 88] Jacob, J.L. "Specifying Security Properties." *(Proceedings of the 1988 Symposium on Security and Privacy)*, IEEE 1988.

[Loc 80] *The InaJo Specification Language Reference Manual.* System Development Corporation, 1980.

[McC 87] McCullough, Daryl "Specifications for Multi–Level Security and a Hook–Up Property." *(Procedings of the 1987 Symposium on Security and Privacy)*, IEEE, 1987.

[Mil 80] Milner, R.A. *A Calculus of Communicating Systems.* Volume 92, LNCS, Springer, 1980.

[Sut 86] Sutherland, D. "A Model of Information." *(Proceedings of the 9th National Computer Security Conference)*, 1986.